EULER : A GENERALIZATION OF ALGOL,

AND ITS FORMAL DEFINITION

BY

NIKLAUS WIRTH and HELMUT WEBER

TECHNICAL REPORT **CS20**

APRIL 27, 1965

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UN IVERS ITY

Technical Report 20

EULER: A Generalization of ALGOL, and its Formal Definition

ERRATA et ADDENDA

p.4,   ℓ4 :   replace "systemizing" by "systematizing" .

       ℓ21:   replace "in [8]" by "here" .

p.5,   ℓ6 :   "standard" should read "fixed" .

p.6,   ℓ14:   underline "productive" .

       ℓ16:   underline "reductive" .

p.10, ℓ21:   replace curly braces { } by parentheses ( ) .

p.11, 110:   dito

       ℓ11:   dito

       ℓ18:   add the following sentence:
              (as an alternative notation for cx we will use x .)

p.13, ℓ3 :   replace {,} by (,) respectively .

P-15, ℓ11:   after "U → x" insert "where" .

       ℓ14:   insert a space after the first z;   ...z (y → z)....

       ℓ17:   dito

p.16, ℓ2 :   underline the word "sentence" .

       ℓ4 :   underline "simple phrase structure language" .

p.26, ℓ9 :   the third symbol to the right of the vertical line should be
              "'"   instead of   "," .

p.37, ℓ17:   change "ennumerate" into "enumerate" .

p .38, ℓ31:   underline the letter V .

       ℓ34: (bottom line) dito.

p.41   : the horizontal line should be between IDENT and DIGIT instead
             of between DIGIT and NUMBER.

P-48, ℓ23:   "a[1]" instead of "a[i]" .

p.52,   ℓ22:   "will" instead of "would" .

p.63, #4,   ℓ6 :   add a semicolon (;) to the right.

              ℓ8 :   dito, also underline label

              ℓ11:   add a semicolon to the right.

p.64, #37,   ℓ2 :   "P[V[j][2] ← k + 1" should be "P[V[j][2]] ← k + 1" .

p.65, #50   :   "isn var" should be "isb var"

p.65, #57   :   change the two occurrences of "isn" into "isu" .

p.67, #114   :   change "blockhead" into "blokhead"

p.70,   ℓ6 :   change the colon at the right into a semicolon.

              ℓ13:   add the symbol "↑" underneath mod .

P-71,   ℓ12:   "At i ← i - 1" should be "A: i ← i - 1" .

p.72,   ℓ4 :   change "string" into "symbol".

              ℓ29:   add a semicolon at the right.

p.73,   ℓ14:   dito

P-75,   ℓ4 :   insert a semicolon in front of "x ← s[1]" .

p.77,   ℓ25:   change "is a number" into "is not a number".

P-91,   ℓ22:   "RESUTS" should read "RESULTS".

P-981   ℓ17:   change "13" at the left into "28".

p.110,   ℓ17:   add to the right: "S[SP].ADR ← FP; COMMENT A NULL LIST;"

2

# EULER : A Generalization of ALGOL, and its Formal Definition*
by
Niklaus Wirth and Helmut Weber

Abstract:

A method for defining programming languages is developed which intro-
duces a rigorous relationship between structure and meaning. The structure
of a language is defined by a phrase structure syntax, the meaning in terms
of the effects which the execution of a sequence of interpretation rules
exerts upon a fixed set of variables, called the Environment. There exists
a one-to-one correspondence between syntactic rules and interpretation rules,
and the sequence of executed interpretation rules is determined by the se-
quence of corresponding syntactic reductions which constitute a parse.
The individual interpretation rules are explained in terms of an elementary
and obvious algorithmic notation. A constructive method for evaluating
a text is provided, and for certain decidable classes of languages their
unambiguity is proven. As an example, a generalization of ALGOL is described
in full detail to demonstrate that concepts like block-structure, procedures,
parameters etc. can be defined adequately and precisely by this method.
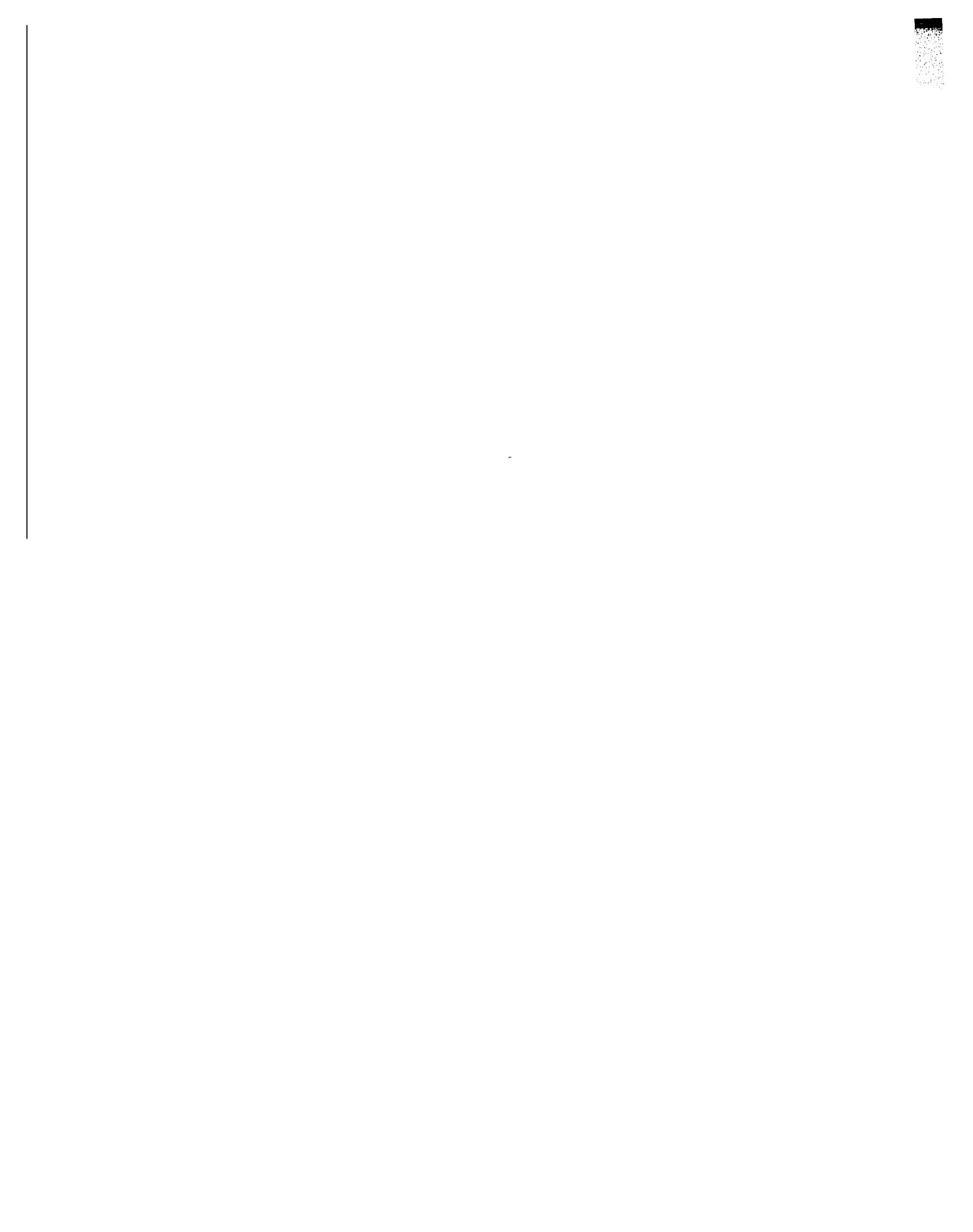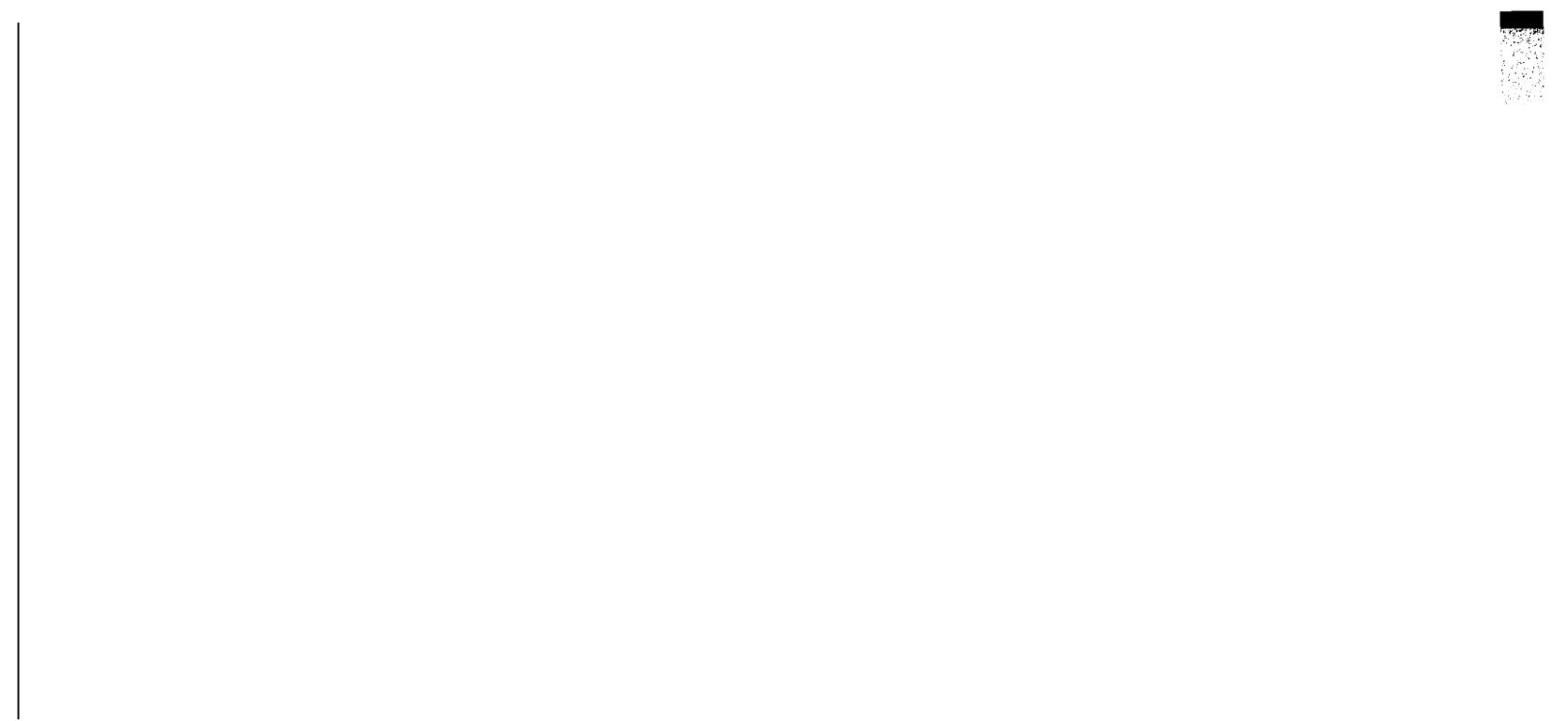

It is the character of mathematics of modern times that through our language of signs and nomenclature we possess a tool whereby the most complicated arguments are reduced to a certain mechanism. Science has thereby gained infinitely, but in beauty and solidity, as the business is usually carried on, has lost so much. How often that tool is applied only mechanically, although the authorization for it in most cases implied certain silent hypotheses! I demand that in all use of calculation, in all uses of concepts, one is to remain always conscious of the original conditions.

Gauss

(in a letter to Schumacher, Sept. 1, 1850)

TABLE OF CONTENTS

I.   Introduction and Summary

When devising a new programming language, one inevitably becomes confronted with the question of how to define it. The necessity of a formal definitiun is twofold: the users of this language need to know its precise meaning, and also need to be assured that the automatic processing systems, i.e. the implementations of the language on computers, reflect this same meaning equally precisely. ALGOL 60 represented the first serious effort to give a formal definition of a programming language [1]. The structure of the language was defined in a formal and concise way (which, however, was not in all cases unambiguous), such that for every string of symbols it can be determined whether it belongs to the language ALGOL 60 or not. The meaning of the sentences, i.e. their effect on the computational pro-cess, was defined in terms of ordinary English with its unavoidable lack of precision. But probably the greater deficiency than certain known im-precise definitions was the incompleteness of the specifications. By this no reference is made to certain intentional omissions (like specifi-cation of _real_ arithmetic), but to situations and constructs which simply were not anticipated and therefore not explained (e.g. dynamic _own_ arrays or conflicts of names upon procedure calls). A method for defining a language should therefore be found which guarantees that no unintentional omissions may occur.

How should meaning be defined? It can only be explained in terms of another language which is already well understood. The method of formally deriving the meaning of one language from another makes sense, if and only if the latter is simpler in structure than the former. By a sequence of such derivations a language will ultimately be reached where it would not

be sensible to define it in terms of anything else.   Recent efforts have been conducted with this principle in mind.

Böhm [3] and Landin [4][5] have chosen the h-calculus as the fundamental notation [6],[7], whose basic element is the function, i.e. a **well**-established concept.   The motivation for representing a program in functional form is to avoid a commitment to a detailed sequence of basic steps representing the algorithm, and instead to define the meaning or effect of a program by the equivalence class of algorithms represented by the indicated function.   Whether it is worth while to achieve such an abstract definition of meaning in the case of programming languages shall not be discussed here.   The fact that a program consists basically of single steps remains, and it cannot even be hidden by a transliteration into a functional notation: the sequence is represented by the evaluations of nests of functions and their parameters.   An unpleasant side-effect of this translation of ordinary programming languages into h-calculus is that simple computer concepts such as assignment and jumps transform into quite complicated constructs, this being in obvious conflict with the stated requirement that the fundamental notation should be simple.

Van Wijingaarden describes in [8] and [9] a more dynamic approach to the problem: the fundamental notation is governed by only half a dozen rules which are obvious.   It is in fact so simple that it is far from being a useful programming notation whatsoever, but just capable enough to provide for the mechanism of accepting additional rules and thus expanding into any desirable programming system.   This method of defining the meaning

2

(or, since the meaning is imperative: effect) of a language is clearly distinct from the method using functional notations, in that it explicitly makes use of algorithmic action, and thus guarantees that an evaluating algorithm exists for any sentence of the language. The essence of this algorithm consists of first scanning the ordered set of rules defining the structure of the language, and determining the applicable structural designations, i.e. performing an 'applicability scan', and then scanning the set of rules for evaluating the determined structural units, i.e. performing an 'evaluation scan'. The rules are such that they may invoke application of other rules or even themselves. The entire mechanism is highly recursive and the question remains, whether a basically subtle and intricate concept such as recursion should be used to explain other programming languages, including possibly very simple ones.

The methods described so far have in common that their basic set of fundamental semantic entities does not resemble the elementary operations performed by any computational device presently known. Since the chief aim of programming languages is their use as communication media with computers, it would seem only natural to use a basic set of semantic definitions closely reflecting the computer's elementary operators. The invaluable advantage of such an approach is that the language definition is itself a processing system and that implementations of the language on actual machines are merely adaptations to particular environmental conditions of the language definition itself. The question of correctness of an implementation will no longer be undecidable or controversial, but can be directly based on the correctness of the individual substitutions of the elementary semantic units by the elementary machine operations.

3

It has elsewhere been proposed (e.g. [10]) to let the processing systems themselves be the definition of the language. Considering the complexity of known compiler-systems this seems to be an unreasonable suggestion, but if it is understood as a call for systemizing such processing systems and representing them in a notation independent from any particular computer, then the suggestion appears in a different light.

The present paper reports on efforts undertaken in this direction. It seems obvious that the definition of the structure, i.e. the syntax, and the definition of the meaning should be interconnected, since structural orderings are merely an aid for understanding a sentence. In the presented proposal the analysis of a sentence proceeds in parallel with its evaluation: whenever a structural unit is discovered, a corresponding interpretation rule is found and obeyed. The syntactic aspects are defined by a Phrase Structure System (cf. [11], [12], [2]) which is augmented by the set of interpretation rules defining the semantic aspects. Such an augmented Phrase Structure Language is subsequently called a Phrase Structure Programming Language, implying that its meaning is strictly imperative and can thus be expressed in terms of a basic algorithmic notation whose constituents are, e.g., the fundamental operations of a computer.

Although in [8] the processes of syntactic analysis and semantic evaluation are more clearly separated, the analogies to the van Wijngaarden proposal are apparent. The parsing corresponds to the applicability scan, the execution of an interpretation rule to the evaluation scan. However, this proposal advocates the strict separation between the rules which define the language, i.e. its analysis and evaluation mechanisms, and the

rules produced by the particular program under evaluation, while the van Wijngaarden proposal does not distinguish between language definition and program. Whether the elimination of this distinction which enables-- and forces--the programmer to supply his own language defining rules, is desirable or not must be left unanswered here. The original aim of this contribution being the development of a proposal for a standard language, it would have been meaningless to eliminate it.

Chapter II contains the descriptions of an algorithmic notation donsidered intuitively obvious enough not to necessitate further expla- nation in terms of more primitive concepts. This notation will subse- quently be used for the definition of algorithms and interpretation rules, thus playing a similar role for the semantic aspects as did BNF for the syntactic aspects of ALGOL 60. The function of this notation is twofold: 1. It serves to precisely describe the analysis and evaluation mechanisms, and 2. It serves to define the basic constituents of the higher level language. E.g., this basic notation contains the elementary operators for arithmetic, and therefore the specifications of the higher level lan- guage defer their definition to the basic algorithmic notation. It is in fact assumed that the definition of <u>integer</u> arithmetic is below the level of what a programming language designer is concerned with, while <u>real</u> arithmetic shall very intentionally not be defined at all in a language standard. The concepts which are missing in the basic notation and thus will have to be defined by the evaluation mechanisms are mani- fold: the sequencing of operations and operands in expressions, the stor- age allocation, the block structure, procedure structure, recursivity, **value-** and name-parameters, etc.

Chapter III starts out with a list of basic formal definitions leading to the terms 'Phrase Structure System' , 'Phrase Structure Programming Language' and 'Meaning' . The notation and terminology of [12] is adopted here as far as possible. The fact that the nature of meaning of a programming language is imperative, allows the meaning of a sentence to be explained in terms of the changes which are affected on a certain set of variables by obeying the sentence. This set of variables is called the Environment of the Programming Language. The definition of the meaning with the aid of the structure, and the definition of the evaluation algorithm in terms of structural analysis of a sentence demand that emphasis be put on the development of a constructive algorithm for a syntactic analysis. Chapter III is mainly devoted to this topic. It could have been entirely avoided, had a reductive instead of a productive definition of the syntax been chosen. By a productive syntactic definition is meant a set of rules illustrating the various constructs which can be generated by a given syntactic entity. By a reductive syntactic definition is meant a set of rules directly illustrating the reductions which apply to a given sentence. A reductive syntax therefore directly describes the analyser, and recently some compilers have been constructed directly relying on a reductive syntactic description of the language. [13]. A language definition, however, is not primarily directed toward the reader (human or artificial), but toward the writer or creative user. His aim is to construct sentences to express certain concepts or ideas. The productive definition allows him to derive directly structural entities which conform to his concepts. In short, his use of the language is primarily synthetic and not analytic in nature. The reader then must apply an analytic process, which

in turn one should be able to specify given the productive syntactic defi-nitions. One might call this a transformation of a productive into a reductive form, a synthetic into an analytic form.

The transformation method derived subsequently is largely based on earlier work by R. W. Floyd described in [14]. The grammars to which this transformation applies are called Precedence Grammars. The term 'Prece-dence Syntax' is, however, redefined, because the class of precedence gram-mars described in [14] was considered to be too restrictive, and even unnec-essarily so. In particular, there is no need to define the class of prece-dence grammars as a subclass of the 'Operator grammars' . Several classes of precedence grammars are defined here, the order of a precedence grammar being determined by the amount of context the analysis has to recognize and memorize in order to make decisions. This classification relates to the definition of 'Structural Connectedness' described in [15], and provides a means to effectively determine the amount of connectedness for a given grammar.

Also in Chapter III, an algorithm is described which decides whether a given grammar is a precedence grammar, and if so, performs the desired transformation into data representing the reductive form of the grammar.

A proof is then provided of the unambiguity of precedence grammars, in the sense that the sequence of syntactic reductions applied to a sen-tence is unique for every sentence in the language. Because the sequence of interpretation rules to be obeyed is determined by the sequence of syntactic reductions, this uniqueness also guarantees the unambiguity of meaning, a crucial property for a programming language. Furthermore, the fact that all possible reductions are described exhaustively by the syntax,

and that to every syntactic rule there exists a corresponding interpretation (semantic) rule, guarantees that the definition of meaning is exhaustive. In other words, every sentence has one and only one meaning, which is well defined, if the sentence belongs to the language. Chapter III ends with a short example: The formal definition of a simple programming language containing expressions, assignment statements, declarations and **block-**structure.

A formal definition of an extension and generalization of ALGOL 60 is presented in Chapter IV. It will demonstrate that the described methods are powerful enough to define adequately and concisely all features of a programming language of the scope of ALGOL 60. This generalization is a further development of earlier work presented in [16].

II. An Elementary Notation for Algorithms.

This notation will in subsequent chapters be used as basis for the definitions of the meaning of more complicated programming languages.

A program is a sequence of imperative statements. In the following paragraphs the forms of a statement written in this elementary notation are defined and rules are given which explain its meaning. There exist two different kinds of statements:

      A.   the <u>Assignment Statement</u>, and

      B.   the <u>Branching Statement</u>.

The Assignment Statement serves to assign a new value to a variable whose old value is thereby lost. The successor of an Assignment Statement is the next statement in the sequence. The Branching Statement serves to designate a successor explicitly. Statements may for this purpose be labelled.

A. The Assignment Statement

The (direct) Assignment Statement is of the form

$$v \leftarrow E \quad .$$

v stands for a <u>variable</u> and E for an <u>expression.</u> The meaning of this statement is that the current value of v is to be replaced by the current value of E.

An expression is a construct of either one of the following forms:

$$x \; , \; \circ \, x \; , \; x \; \Theta \; y \; , \; r$$

where **x, y,** stand for either <u>variable</u>s, <u>literals</u> or <u>lists</u>, o stands for a <u>unary operator</u>, Θ stands for a <u>binary operator</u> and **r** stands for a <u>reference.</u> The value of an expression involving an operator is obtained by applying the operator to the current value(s) of the operand(s).

A reference is written as @ **v,** where v is the referenced variable.

The <u>indirect</u> Assignment Statement is written as

$$\mathbf{v.} \leftarrow \mathbf{E}$$

and is meant to assign the current value of the expression E to the variable, whose reference is currently assigned to the variable **v** .

1. <u>Literals</u>

A literal is an entity characterized by the property that its value is always the literal itself. There may exist several kinds of literals, e.g.'

        Numbers

        Logical constants (Boolean)

        Symbols

Furthermore there exists the literal Ω with the meaning "undefined". Numeric constants shall be denoted in standard decimal form. The logical constants are <u>true</u> and <u>false</u>*.

A symbol or character is denoted by the symbol itself enclosed in quote marks (**''**). A list of **symbols** is usually called a **string**. Other types of literals may arbitrarily be introduced.

2. <u>Lists</u>

A list **is an** entity denoted by'

$$\{\mathbf{E, F, \ . \ . \ ., G}\}$$

whose value is the ordered set of the current values of the expressions E, F, . . . , G, called the elements of the list. A list can have any **number** of elements (including 0), and the elements are numbered with the natural numbers starting with 1 .

_____

*   the underlined (boldface) letters have to be understood as one single symbol.

3. Variables

A variable is an entity uniquely identified within a program by a name to which a value can be assigned (and reassigned) during the execution of a program. Before the first assignment to a variable, its value shall be Ω .

If the value of a variable consists of a sequence of elements, any one element may be designated by the variable name and a subscript, and thus is called a subscripted variable. The subscript is an expression, whose current value is the ordinal number of the element to be designated. Thus, after a ← {1,2,{3,4,5,},6}, a[1] designates the element "1", a[3] designates the element {3,4,5}, and therefore a[3][2] designates the second element of a[3], i.e. "4". The notation a_i shall be understood equivalent to a[i], $a_{i,j}$ equivalent to a[i][j] etc.

4. Unary Operators

Examples of unary operators are:

- **x** , yields the negative of x

<u>c</u> x , yields the value of the variable whose reference is currently assigned to x

<u>abs</u> x , yields the absolute value of x

<u>integer</u> x , yields x rounded to the nearest integer

<u>tail</u> x , yields the list x with its first element deleted;

<u>isli</u> x , yields <u>true</u>, if x <u>is</u> a <u>list</u>, <u>false</u> otherwise

A further set of unary operators is the set of **typetest** operators which determine whether the current value of a variable is a member of a certain set of literals. The resulting value is <u>true,</u> if the test is affirmative, <u>false</u> otherwise.

11

Examples:

    <u>isn</u> x, current value of x <u>is</u> a <u>n</u>umber

    <u>isb</u> x, ...is a logical (Boolean) constant

    <u>isu</u> x, ...is Ω (undefined)

    <u>isy</u> x, ...is a <u>s</u>ymbol

A further set of unary operators is the set of conversion operators which produce values of a certain type from a value of another type:

Examples:

    <u>real</u> x yields the number corresponding to the logical value x;

  <u>logical</u> x inverse of <u>real</u> (<u>true</u> ↔ 1, <u>false</u> ↔ 0 shall be assumed);

           Conversion operators between numbers and symbols shall not

           be defined here, although their existence is assumed, because

           the notation does not define the set of symbols which may

           possibly be used.

5. <u>Binary Operators</u>

Examples of binary operators are:

+ - X designating addition, subtraction and multiplication in the usual sense. The accuracy of the result in the case of the operands being non-integral numbers is not defined.

/   denoting division in the usual sense. The accuracy of the result is not defined here. In case of the denominator being 0, the result is Ω .

÷   denoting division between the rounded operands with the result being truncated to its integral value.

12

<u>mod</u>   yields the remainder of the division denoted by ÷ .

&   yields the concatenation of two lists, i.e.

$$\{x\} \ \& \ \{y\} = \{x,y\}$$

=   yields <u>true,</u> if the two scalar operands are equal, <u>false</u>

otherwise.

↑   denoting exponentiation, i.e. $x \uparrow y$ stands for x^y .

The classes of unary and binary operators listed here may be ex-
tended and new types of literals may be introduced along with corresponding
typetest and conversion operators.

B. <u>The Branching Statement</u>

There are Simple and Conditional Branching Statements.

1. <u>The Simple Branching Statement</u>

It is of the form

<u>goto</u> ℓ

where ℓ stands for a label. The meaning is that the successor

of this statement is the statement with the label ℓ . Labelling

of a statement is achieved by preceding it with the label and a

colon (:). The label is a unique name (within a program) and desig-

nates exactly one statement of the program.

2. <u>The Conditional Branching Statement</u>

It is of the form

if E <u>then goto</u> ℓ

where ℓ is a label uniquely defined in the program and E is

an expression. The meaning is to select as the successor to the

13

Branching Statement the statement with the label l, if the current

value of E is t̲r̲u̲e̲, or the next statement in the sequence, if it

is f̲a̲l̲s̲e̲. For notational convenience a statement of the form

$$i̲f̲ \; \neg \; E \; t̲h̲en \; got̲o̲ \; l \qquad (\neg = not)$$

shall be admitted and understood in the obvious sense.

****

Notational standards shall not be fixed here. Thus the sequence

of statements can be established by separating statements by delimiters,

or by beginning a new line for every statement. The Branching Statement

and the labelling of statements may be replaced by explicit arrows, thus

yielding block diagrams or flow-charts.

III. Phrase Structure Programming' Languages'!,'

A. Notation, Terminology, Basic Definitions

Let $\mathcal{V}$ be a given set: the vocabulary. Elements of $\mathcal{V}$ are called symbols and will be denoted by capital Latin letters, S, T, U etc. Finite sequences of symbols -- including the empty sequence ($\wedge$) -- are called strings and will be denoted by small Latin letters -- x, y, z, etc. The set of all strings over $\mathcal{V}$ is denoted by $\mathcal{V}*$. Clearly $\mathcal{V} \subseteq \mathcal{V}^*$.

A simple phrase structure system is an ordered pair $(\mathcal{V}, \Phi)$, where $\mathcal{V}$ is a vocabulary and Φ is a finite set of syntactic rules φ of the form

$$U \to x \quad \neq \quad U \in \mathcal{V}, x \in \mathcal{V}^*) .$$

For $\varphi = U \to x$, U is called the left part and x the right part of φ.

y directly produces $z(y \overset{.}{\to} z)$ and conversely z directly reduces into y, if and only if there exist strings u, v such that y = uUv and z = uxv, and the rule $U \to x$ is an element of Φ .

y produces $z(y \overset{*}{\to} z)$ and conversely z reduces into y, if and only if there exist a sequence of strings $x_0, \ldots, x_n$, such that y = x_0, x_n = z, and $x_{i-1} \overset{.}{\to} x_i$ (i-= 1,...,n;n > 1) .

A simple phrase structure syntax is an ordered quadruple $\mathcal{G} = (\mathcal{V}, \Phi, \mathcal{B}, A)$, where $\mathcal{V}$ and Φ form a phrase structure system; $\mathcal{B}$ is the subset of $\mathcal{V}$ such that none of the elements of $\mathcal{B}$ (called basic symbols) occurs as the left part of any rule of Φ, while all elements of $\mathcal{V}-\mathcal{B}$ occur as left part of at least one rule; A is the symbol which occurs in no right part of any rule of Φ .

15

The letter U shall always denote some symbol $U \in \mathcal{V}-\mathcal{B}$.

x is a sentence of $\mathcal{G}$, if $x \in \mathcal{V}^*$ (i.e. x is a string of basic symbols) and $A \xrightarrow{*} x$.

A simple phrase structure language $\mathcal{L}$ is the set of all strings x which can be produced by $(\mathcal{V}, \Phi)$ from A:

$$\mathcal{L}(\mathcal{G}) = \{x | A \xrightarrow{*} x \wedge x \in \mathcal{V}^*\} \ .$$

Let U 3 z . A parse of the string z into the symbol U is a sequence of syntactic rules $\varphi_1, \varphi_2, \ldots \varphi_n$, such that φ_j directly reduces z_{j-1} into zj (j = 1 . . . n), and $z = z_0$, $z_n = U$.

Assume $z_k = U_1 U_2 \ldots U_m$ (for some $1 < k < n$) . Then z_i (i < k) must be of the form $z_i = u_1 u_2 . _* . u_m$, where for each $\ell = 1 \ldots m$ either $U_\ell \xrightarrow{*} u_\ell$, or $U_\ell = u_\ell$. Then the canonical form of the section of the parse reducing z_i into z_k shall be $\{\varphi_1\}\{\varphi_2\} \ldots \{\varphi_m\}$, where the sequence $\{\varphi_\ell\}$ is the canonical form of the section of the parse reducing u_ℓ into U_ℓ . Clearly $\{\varphi_\ell\}$ is empty, if $U_\ell = u_\ell$, and is canonical, if it consists of 1 element only..

The canonical parse is the parse which proceeds strictly from left to right in a sentence, and reduces a leftmost part of a sentence as far as possible before -proceeding further to the right. In general, there may exist several canonical parses for a sentence, but every parse has only one canonical form.

An unambiguous syntax is a phrase structure syntax with the property that for every string $x \in \mathcal{L}(\mathcal{G})$ there exists exactly one canonical parse.

It has been show-n that there exists no algorithm which decides the ambiguity problem for any arbitrary syntax. However, a sufficient condition for a syntax to be unambiguous will subsequently be derived. A method will be explained to determine whether a given syntax satisfies

this condition.

An environment $\mathcal{E}$ is a set of variables whose values define the meaning of a sentence.

An interpretation rule ψ defines an action (or a sequence of actions) involving the variables of an environment $\mathcal{E}$.

A phrase structure programming language $\mathcal{L}_p(\mathcal{G}, \Psi, \mathcal{E})$ is a phrase structure language $\mathcal{L}(\mathcal{G})$, where $\mathcal{G}(\mathcal{V}, \Phi, \mathcal{B}, A)$ is a phrase structure syntax, Ψ is a set of (possibly empty) interpretation rules such that a'unique one to one mapping exists between elements of Ψ and Φ, and $\mathcal{E}$ is an environment for the elements of Ψ. Instead of $\mathcal{L}_p(\mathcal{G}, \Psi, \mathcal{E})$ we also write $\mathcal{L}_p(\mathcal{V}, \Phi, \mathcal{B}, A, \Psi, \mathcal{E})$.

The meaning m of a sentence $x \in \mathcal{L}_p$ is the effect of the execution of the sequence of interpretation rules $\psi_1, \psi_2 \ldots \psi_n$ on the environment $\mathcal{E}$, where $\varphi_1 \varphi_2 \ldots \varphi_n$ is a parse of the sentence x into the symbol A and ψ_i corresponds to φ_i for all i.

It follows immediately that a programming language will have an unambiguous meaning, if and only if its underlying syntax is unambiguous. As a consequence, every sentence of the language has a well-defined meaning.

A sentence $x_1 \in \mathcal{L}_p(\mathcal{G}_1, \Psi_1, \mathcal{E})$ is called equivalent to a sentence $x_2 \in \mathcal{L}_p(\mathcal{G}_2, \Psi_2, \mathcal{E})$ (possibly $\mathcal{G}_1 = \mathcal{G}_2$, $\Psi_1 = \Psi_2$), if and only if $m(x_1)$ is equal to $m(x_2)$.

A programming language $\mathcal{L}_p(\mathcal{G}_1, \Psi_1, \mathcal{E})$ is called equivalent to $\mathcal{L}_p(\mathcal{G}_2, \Psi_2, \mathcal{E})$, if and only if $\mathcal{L}_{p1} = \mathcal{L}_{p2}$ and for every sentence x, $m_1(x)$ according to $(\mathcal{G}_1, \Psi_1)$ is equal to $m_1(x)$ according to $(\mathcal{G}_2, \Psi_2)$.

B. Precedence Phrase Structure Systems

The definition of the meaning of a sentence requires that a sentence must be parsed in order to be evaluated or obeyed. Our prime attention will therefore be directed toward a constructive method for parsing. In the present chapter, a parsing algorithm will be described. It relies on certain relations between symbols. These relations can be determined for any given syntax. A syntax for which the relation between any two symbols is unique, is called a simple precedence syntax. Obviously, the, parsing algorithm only applies to precedence phrase structure systems. It will then be shown that any parse in-such a system is unique. The class of precedence phrase structure systems is only a restricted subset among all phrase structure systems. The definition of precedence relations will subsequently be generalized with the effect that the class of precedence phrase structure systems will be considerably enlarged.

1. The Parsing Algorithm for Simple Precedence Phrase Structure Languages.

In accordance with the definition of the canonical form of a generation tree or of a parse, a parsing algorithm must first detect the leftmost substring of the sentence to which a reduction is applicable. Then the reduction is to be performed and the same principle is applied to the new sentence. In order to detect the leftmost reducible substring, the algorithm to be presented here makes use of previously established noncommutative relations between symbols of $\mathcal{V}$ which are chosen according to the following criteria:

a. The relation $\doteq$ holds between all adjacent symbols within a string which is directly reducible;

18

b. The relation $\lessdot$ holds between the symbol immediately pre-

ceding a reducible string and the leftmost symbol of that string;

c. The relation $\gtrdot$ holds between the rightmost symbol of a

reducible string and the symbol immediately following that string.

The process of detecting the leftmost reducible substring now consists

of scanning the sentence from left to right until the first symbol

pair is found so that $S_i \gtrdot S_{i+1}$, then to retreat back to the last

symbol pair for which $S_{j-1} \lessdot S_j$ holds. $S_j \ldots S_i$ is then the

sought substring; it is replaced by the symbol resulting from the

reduction. The process then repeats itself. At this point it must

be noted that it is not necessary to start scanning at the beginning

of the sentence, since all symbols S_k for $k < j$ have not been

altered, but that the search for the next $\gtrdot$ can start at the place

of the previous reduction.

In the following formal description of the algorithm the original

sentence is denoted by $P_1 \ldots P_n$. k is the index of the last symbol

scanned. For practical reasons, all scanned symbols are copied and

renamed $S_j \ldots S_i$. The reducible substring therefore will always be

$S_1 \ldots S_i$ for some j . Internal to the algorithm, there exists a

symbol $\bot$ initializing and terminating the process. To any symbol

S of ϑ it has the relations $- \bot \lessdot S$ and $S \gtrdot \bot$.

We assume that $P_0 = P_{n+1} = \bot$.

START

i ← 1
k ← 1
$S_1 ← \perp$

i ← i + 1
j ← i
$S_i ← P_k$
k ← k + 1

①

$S_i = \perp$ → Y → STOP

N

$S_i > P_k$ → N

Y

③

②

$S_{j-1} \doteq S_j$ → Y → j←j−1

N

Reduce
$S_j \ldots S_i$
i c j
$S_i ← U$

Algorithm for Syntactic Analysis

20
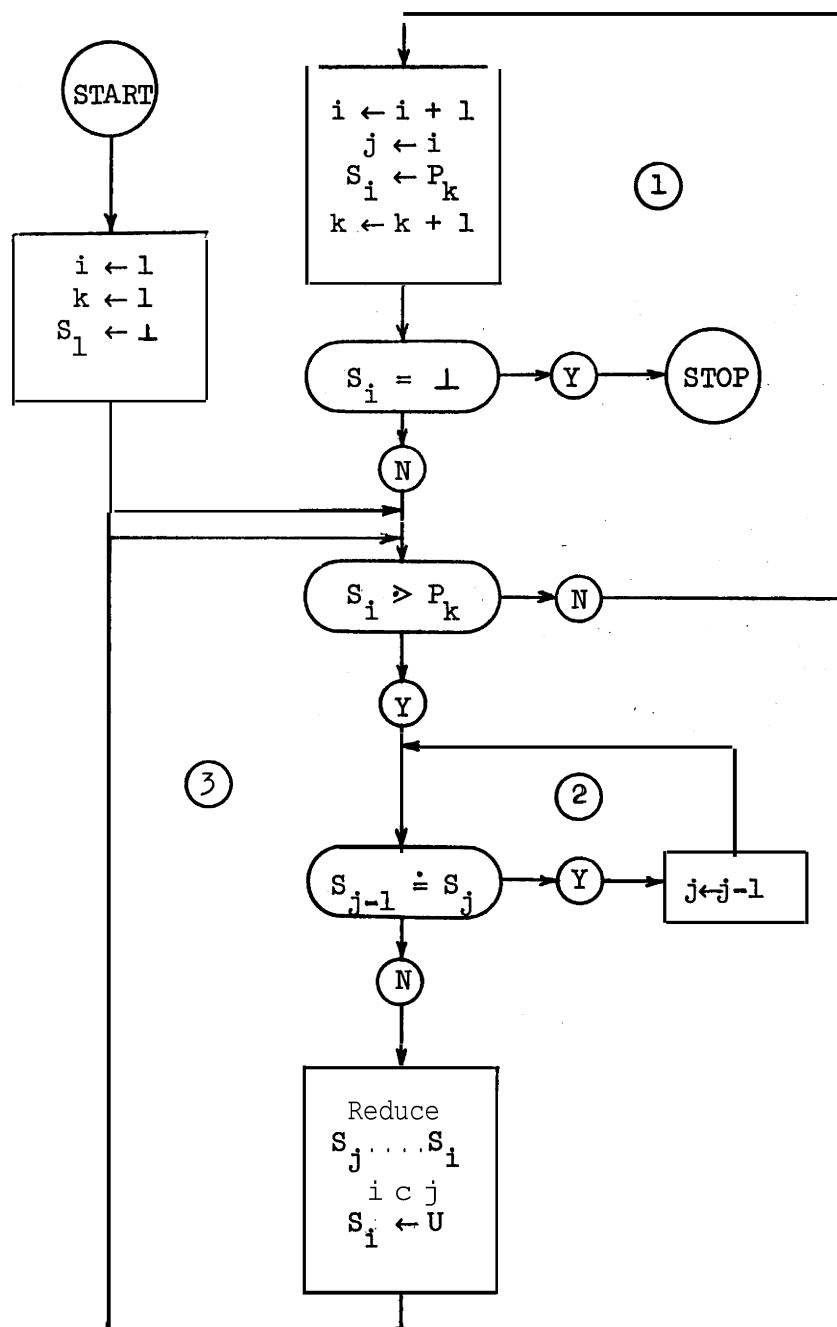
Comments to the Algorithm:

① Copy the string P into S and advance until a relation $\gtrdot$ is

encountered;

② Retreat backward across the reducible substring;

③ A reduction has been made. Resume the search for $\gtrdot$.

The step denoted by "Reduce $S_j \ldots S_i$" requires that the reducible

substring is identified in order to obtain the symbol resulting from

the reduction. If the parsed sentence is to be evaluated, then the

interpretation rule ψ_ℓ corresponding to the syntactic rule φ_ℓ:

$u \rightarrow S_j \ldots S_i$ is identified and obeyed.

2. <u>An Algorithm to Determine the Precedence Relations</u>.

The definition of the precedence relations can be formalized in

the following way:

    a. For any ordered pair of symbols (S_i, S_j), $S_i \doteq S_j$, if and only

        if there exists a syntactic rule of the form $u \rightarrow xS_i S_j y$,

        for some symbol U and some (possibly empty) strings x, y.

    b. For any ordered pair of symbols (S_i, S_j), $S_i \lessdot S_j$, if and only

        if there exists a syntactic rule of the form $u \rightarrow xS_i U_\ell y$,

        for some U, x, y, U_ℓ, and there exists a generation

        $U_\ell \overset{*}{\rightarrow} S_j z$, for some string z.

    c. For any ordered pair of symbols (S_i, S_j), $S_i \gtrdot S_j$, if and

        only if

        1. there exists a syntactic rule of the form $U \rightarrow xU_k S_j y$,

            for some U, x, y, U_k, and there exists a generation

            $U_k \overset{*}{\rightarrow} zs_i$ for some string z, or

2. there exists a syntactic rule of the form $U \to xU_kU_\ell y$,

for some U, x, y, U_k, U_ℓ, and there exist generations

$U_k \overset{*}{\to} zS_i$ and $U_\ell \overset{*}{\to} S_j w$ for some strings z, w .

We now introduce the sets of leftmost and rightmost symbols of a non-basic symbol U by the following definitions:

$$\mathscr{L}(U) = \{S \mid \exists z(U \overset{*}{\to} Sz)\}$$

$$\mathscr{R}(U) = \{S \mid \exists z(U \overset{*}{\to} zS)\}$$

Now the definitions a. b. c. can be reformulated as:

a. $S_i \doteq S_j \longleftrightarrow \exists\varphi(\varphi: U \to xS_iS_jy)$

b. $S_i \lessdot S_j \; c\text{-}3 \; \exists\varphi(\varphi: U \to xS_iU_\ell y) \wedge S_j \in \mathscr{L}(U_\ell)$

c. $S_i \gtrdot S_j \longleftrightarrow \exists\varphi(\varphi: U \to xU_kS_jy) \wedge S_i \in \mathscr{R}(U_k) \vee$

$\exists\varphi(\varphi: U \to xU_kU_\ell y) \wedge S_i \in \mathscr{R}(U_k) \wedge S_j \in \mathscr{L}(U_\ell)$

These definitions are equivalent to the definitions of the precedence relations, if Φ does not contain any rules of the form $u \to \Lambda$, where Λ denotes the empty string.

The definition of the sets $\mathscr{L}$ and $\mathscr{R}$ is such that an algorithm for effectively creating the sets is evident. A symbol S is a member of $\mathscr{L}(U)$, if

a. There exists a syntactic rule $\varphi: U \ni Sx$, for some x, or

b. There exists a syntactic rule $\varphi: u \to U_1x$, and $S \in \mathscr{L}(U_1)$;

i.e.

$$\mathscr{L}(U) . \{S \mid \exists\varphi: . \to . \_ \vee \exists\varphi: . \to U_1x \wedge S \in \mathscr{L}(U_1)\} \quad .$$

Analogously:

$$\mathscr{R}(U) . \{S \mid \exists\varphi: U \to xS \vee \exists\varphi: U \to xU_1 \wedge S \in \mathscr{R}(U_1)\} \quad .$$

22

The algorithm for finding $\mathcal{L}$ and $\mathcal{R}$ for all symbols $U \in \mathcal{V}\text{-}\mathcal{B}$ involves searching Φ for appropriate syntactic rules. In practice, this turns out to be a rather intricate affair, because precautions must be taken when recursive definitions are used. An algorithm is presented in Appendix I as part of an Extended ALGOL program for the Burroughs B5500 computer.

The precedence relations can be represented by a matrix $\underline{M}$ with elements $\underline{M}_{ij}$ representing the relation between the ordered symbol pair (S_i, S_j). The matrix clearly has as many rows and columns as there are symbols in the vocabulary $\mathcal{V}$.

Assuming that an arbitrary ordering of the symbols of $\mathcal{V}$ has been made $(\mathcal{V} = \{S_1, S_2, \ldots, S_n\})$, an <u>algorithm</u> for the determination of the <u>precedence matrix</u> M can be indicated as follows:

For every element φ of Φ which is of the form

$$U \rightarrow S_1 \, S_2 \, \ldots \, S_m$$

and for every pair S_i, S_{i+1} $(i = 1 \ldots m - 1)$ assign

a. $\doteq$ to $\underline{M}_{i, i+1}$;

b. $\lessdot$ to all $\underline{M}_{i, k}$ with row index k such that $S_k \in \mathcal{L}(S_{i+1})$;

c. $\gtrdot$ to all $\underline{M}_{k, i+1}$ with column index k such that $S_k \in \mathcal{R}(S_i)$;

d. $\bullet >$ to all $\underline{M}_{\ell, k}$ with indices ℓ, k such that $S_\ell \in \mathcal{R}(S_i)$ and $S_k \in \mathcal{L}(S_{i+1})$.

Assignments under b. occur only if $S_{i+1} \in \mathcal{V}\text{-}\mathcal{B}$, under c. only if $S_i \in \mathcal{V}\text{-}\mathcal{B}$, and under d. only if both $S_i, S_{i+1} \in \mathcal{V}\text{-}\mathcal{B}$, because $\mathcal{L}(S)$ and $\mathcal{R}(S)$ are empty sets for all $S \in \mathcal{B}$.
This algorithm appears as part of the ALGOL program listed in Appendix I.

A syntax is a <u>simple precedence syntax,</u> if and only if at most one relation holds between any ordered pair of symbols.

3. Examples

a. $\mathcal{G}_1 = (\mathcal{V}_1, \Phi_1, \mathcal{B}_1, S)$

$\mathcal{V}_1 = \{S, H, \lambda, "\}$

$\mathcal{B}_1 = CA, "\}$

Φ_1 :

$$S \rightarrow H\ "$$
$$\underline{\overline{H}} \rightarrow\ "$$
$$H \rightarrow H\ \lambda$$
$$H \rightarrow H\ S$$

Assume that S stands for 'string' and H for 'head', then this phrase structure system would define a string as consisting of a sequence of string elements enclosed in quote-marks, where an element is either λ or another (nested) string.

U	$\mathcal{L}(U)$	$\mathcal{R}(U)$
S	" H	"
H	" H	" λ S

M	S	H	λ	"
S	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
H	$\doteq$	$\lessdot$	$\doteq$	$\lessdot\!\!\doteq$
λ	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
"	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$

Since both $H \doteq "$ and $H \lessdot "$, $\mathcal{G}_1$ is not a precedence syntax. It is intuitively clear that either nested strings should be delineated by distinct opening and closing marks $(\mathcal{G}_2)$, or that no nested strings should be allowed $(\mathcal{G}_3)$.

b. $G_2 = (V_2, \Phi_2, B_2, S)$

$V_2 = \{S, H, \lambda, ', '\}$

$B_2 = \{\lambda, ', '\}$

Φ_2 : S → H' (φ_1)
         H → ' (φ_2)
         H → Hλ (φ_3)
         H → HS (φ_4)

U	$L(U)$	$R(U)$
S	' H	' λ S '
H	' H	' λ S '

$\underline{M}$	S	H	λ	'	'
S	⋗	⋗	⋗	≐	⋗
H	≐	⋗	≐	⋗	≐
λ	⋗	⋗	⋗	⋗	⋗
'	⋗	⋗	⋗	⋗	⋗
'	⋗	⋗	⋗	⋗	⋗

G_2 is a precedence syntax

c. $G_3 = (V_1, \Phi_3, B_1, S)$

Φ_3 : S → H"
         H → "
         H → Hλ

U	$L(U)$	$R(U)$
S	" H	" H
H	" H	" λ

M	S	H	λ	"
S				
H			≐	≐
λ			3	⪢
"			⪢	⪢

$\mathcal{G}_3$ is a precedence syntax.

As an illustration for the parsing algorithm, we choose the parsing
of a sentence of $\mathcal{L}(\mathcal{G}_2)$:
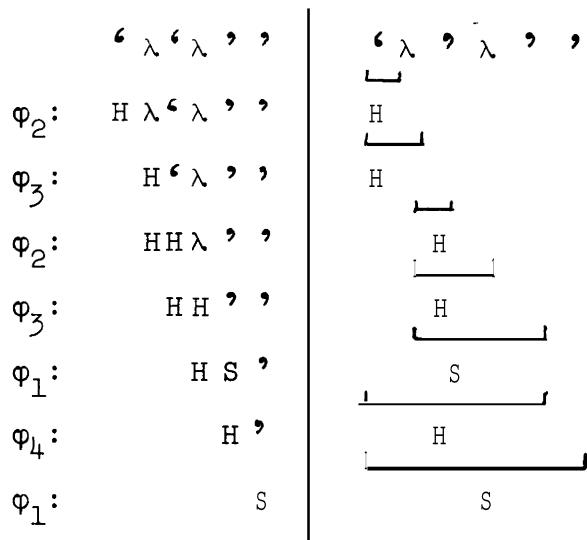
4. The Uniqueness of a Parse.

The three previous examples suggest that the property of unique
precedence relationship between all symbol pairs be connected with unique-
ness of a parse for any sentence of a language. This relationship is
established by the following theorem:

Theorem: The given parsing algorithm yields the canonical form of the
parse for any sentence of a precedence phrase structure language, if there
exist no two syntactic rules with the same right part. Furthermore, this
canonical parse is unique.

This theorem is proven, if it can be shown that in any sentence its directly reducible parts are disjoint. Then the algorithm, proceeding strictly from left to right, produces the canonical parse, which is unique, because no reducible substring can apply to more than one syntactic rule.

The _proof_ that all directly reducible substrings are disjoint is achieved indirectly: Suppose that the string $S_1 \ldots S_n$ contain two directly reducible substrings $S_1 \ldots Sk$ (a.) and $S_{ij} \ldots S_\ell$ (b.), where $1 \leq i \leq j \leq k \leq \ell \leq n$. Then because of a. it follows from the definition of the precedence relations that $S_{j-1} \doteq S_j$ and $S_k \gtrdot S_{k+1}$, and because of b. $S_{j-1} \lessdot S_j$ and $S_k \doteq S_{k+1}$. Therefore this sentence cannot belong to a precedence grammar.

Since in particular the leftmost reducible substring is unique, the syntactic rule to be applied is unique. Because the new sentence again belongs to the precedence language, the next reduction is unique again. It can be shown by induction, that therefore the entire parse must be unique.

From the definition of the meaning of a phrase structure programming language it follows that its meaning is unambiguous for all sentences, if the underlying syntax is a precedence syntax.

5. Precedence Functions.

The given parsing algorithm refers to a matrix of precedence relations with n^2 elements, where n is the number of symbols in the language. For practicalcompilers this would in most cases require an extensive amount of storage space. Often the precedence relations are such that two numeric functions (f, g) ranging over the set of symbols can

be found, such that for all ordered pairs (S_i, S_j)

a. $f(S_i) = g(S_j) \longleftrightarrow S_i \doteq S_j$

b. $f(S_i) < g(S_j) \longleftrightarrow S_i \lessdot S_j$

c. $f(S_i) > g(S_j) \longleftrightarrow S_i \gtrdot S_j$

If these functions exist and the parsing algorithm is adjusted appropriately, then the amount of elements needed to represent the precedence information reduces from n^2 to $2n$. An algorithm for deciding whether the functions exist and for finding the functions if they exist is given as part of the ALGOL program in Appendix-1 .

In example $\mathcal{G}_2$ e.g. the precedence matrix can be represented by the two functions f and g, where

S =	s	h	λ	'	'
$f(S)$ =	3	1	3	3	3
$g(s)$ =	1	2	1	2	1

A precedence phrase structure syntax for which these precedence functions do not exist is given presently:

$\mathcal{V} = \{A, B, C, \lambda, [,]\}$

$\mathcal{B} = \{\lambda, [,]\}$

$\Phi:$ $A \rightarrow C\ B\ ]$

       $A \rightarrow [\ ]$

       $B \rightarrow \lambda$

       $B \rightarrow \lambda\ A$

       $B \rightarrow A$

       $C \rightarrow [$

It can be verified that this is a precedence syntax and in particular the following precedence relations can be derived:

$$\lambda \lessdot [\ , \ [\ \gtrdot \ [\ , \ [\ \doteq \ ] \ , \ \lambda \gtrdot \ ]$$

Precedence functions f and g would thus have to satisfy

$$f(\lambda) < g([) < f([) = g(]) < f(\lambda)$$

which clearly is a contradiction. Precedence functions therefore do not
exist for this precedence syntax.

6. Higher Order Precedence Syntax.

It is the purpose of this chapter to redefine the precedence
ralationships more generally, thus enlarging the class of precedence phrase
structure systems. This is desirable, since for precedence languages a
constructive parsing algorithm has been presented which is instrumental
in the definition of the meaning of the language. The motivation for the
manner in which the precedence relationships will be generalized is first
illustrated in an informal way by means of examples. These examples are
phrase structure systems which for one or another reason might be likely
to cccur in the definition of a language, but which also violate the rules
for simple precedence syntax.

Example 1.

$\mathcal{V}$ = (A , B , ; , S , D}

$\mathcal{B}$ = {; , S , D}

Φ: A → B
  A → D ; A
  B → S
  B → B ; S

S ∈ $(A), thus ; $\lessdot$ S ,   and also ; $\doteq$ S .

This syntax produces sequences of D's separated by ";", followed
by a sequence of symbols S, also separated by ";" . A parse is con-
structed as follows:

```
D ; ...... ; D ; D ; S ; S ; ...,.. ;' S
                        ⊔
                        B
                    ⌊___⌋
                        B
                    . . . . . . . . . . . . .
                    ⌊_____⌋
                            B
                    ⌊_____⌋
                            A
                ⌊_____⌋
                            A
            ⌊_____⌋
                        A            ** a
            . . . . . . . . . . . . . . . . . .
        ⌊_____⌋
                        A
```
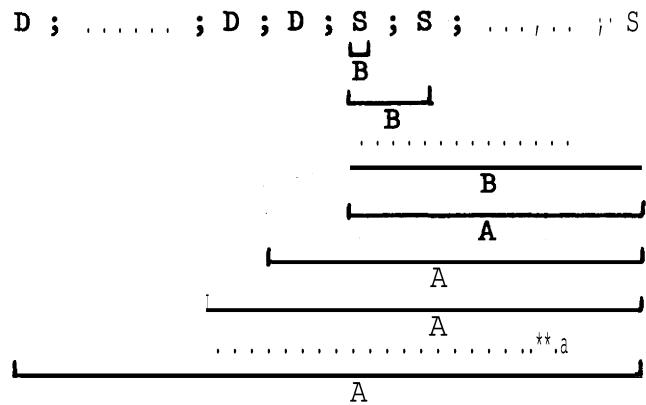
The sequence of S's is defined using a left-recursive definition. while

the sequence of D's is defined using a right-recursive definition. The

precedence violation occurs, because for both sequences the same separator

symbol is used.

The difficulty **arises** when the symbol sequence   " **;S** " occurs. It

is then not clear whether both symbols should be included in the same **sub-**

string or not.   The decision can be made, if the immediately <u>preceding</u>

symbol is investigated.

In other words, not only two single symbols should be related, but a

symbol and the string consisting of the two previously obtained symbols.

Thus:

$$B; \doteq S \quad \text{and} \quad D; < S \ .$$

<u>Example 2:</u>

$\mho = [A , B , ; , S , D),$

$\mathcal{B} = \{; , S , D\}$

$\Phi: \quad A \to B$
$\quad \quad A \to A \ ; \ S$
$\quad \quad B \to D$
$\quad \quad B \to D \ ; \ B$

30

D $\in$ R(A), thus D $\gtrdot$; and also D $\doteq$;

This syntax produces the same strings as the preceding one, but with a different syntactic structure:

$$\mathbf{D} \; ; \; \ldots \ldots \; ; \; D \; ; \; \underbrace{D}_{B} \; ; \; S \; ; \; S \; ; \; \ldots \ldots \; ; \; S$$
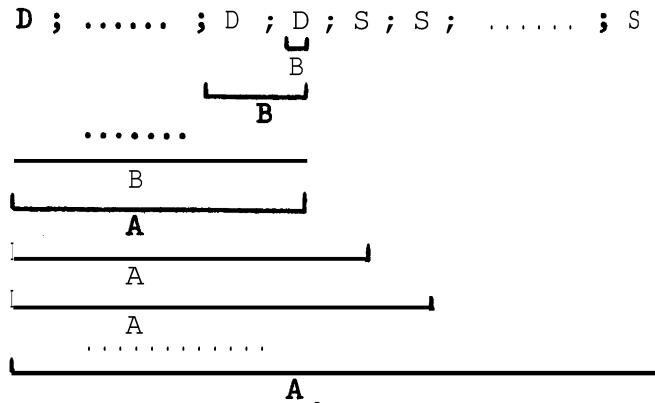
Here the same difficulty arises upon encountering the symbol sequence "**D;**" . The decision whether to include both symbols in the same syntactic category or not can be reached upon investigating the <u>following</u> symbol. Explicitly, a symbol should be related to the subsequent string of 2 symbols, i.e.

$$\mathbf{D} \doteq \; ;D \quad \text{and} \quad \mathbf{D} \gtrdot \; ;\mathbf{S} \; .$$

<u>Example 3</u>:

$\mho = \{A \; , \; B \; , \; \lambda \; , \; ; \; , \; [\; , \;]\}$

$\mathcal{B} = \{\lambda \; , \; ; \; , \; [\; , \;]\}$

$\Phi \; : \; A \to B \; ; \; B$
$ B \to [\; A \;]$
$ B \to [\; \lambda \;]$
$ B \to \lambda$

Since $\lambda \in \mathcal{L}(A)$ and $A \in$ %(A) : [$\lessdot \lambda$ and $\lambda \gtrdot$] . But 'also [$\doteq \lambda$ and $\lambda \doteq$] .

In this case the following relations must be established to resolve the ambiguity.

$$[\doteq \lambda] \; , \quad [\lessdot \lambda ; \; , \quad ;\lambda \gtrdot] \quad \text{and} \quad [\lambda \doteq] \quad .$$

31

This syntax therefore combines the situations arising in Examples 1 and 2. Obviously, examples could be created where the strings to be related would be of length greater than 2. We will therefore call a precedence phrase structure system to be of order (m, n), if unique precedence relations can be established between strings of length $\leq m$ and strings of length $\leq n$. Subsequently, a more precise definition will be stated. A set of extended rules must be found which define the generalized precedence relations. The parsing algorithm, however, remains the same, with the exception that not only the symbols S_i and P_k be related, but possibly the strings $S_{i_m} \ldots S_i$ and $P_k \ldots P_{k+n}$.

The definitions of the relations $\lessdot \doteq, \gtrdot$ is as follows: Let $x = S_{-m} \cdot \ldots S_{-1}$, $y = S_1 \ldots S_n$, let $u, v, u', v' \in \mathcal{V}^*$ and $U, U_1, U_2 \in \mathcal{V}-\mathcal{B}$, then

a. $\mathbf{x} \doteq \mathbf{y}$, if and only if there exists a syntactic rule

$u \rightarrow uS_{-1}S_1 v$, and

$uS_{-1} \overset{*}{\rightarrow} u'x, \quad S_1 v \overset{*}{\rightarrow} y\text{-}v'$;

b. $\mathbf{x} \lessdot \mathbf{y}$, if and only if there exists a syntactic rule

$u \rightarrow uS_{-1}U_1 v$, and

$uS_{-1} \overset{*}{\rightarrow} u'x, U_1 v \overset{*}{\rightarrow} yv'$;

c. $\mathbf{x} \gtrdot \mathbf{y}$, if and only if there exists a syntactic rule

$U \rightarrow uU_1 S_1 v$, and

$uU_1 \overset{*}{\rightarrow} u'x, \quad S_1 v \overset{*}{\rightarrow} yv'$, or there exists a syntactic rule

$U \rightarrow uU_1 U_2 v$ and $uU_1 \overset{*}{\rightarrow} u'x, \quad U_2 v \overset{*}{\rightarrow} yv'$.

A syntax is said to be a precedence syntax of order (m,n), if and only if

    a.   it is not a precedence syntax of degree (m', n') for $m' < m$

        or $n' < n$, and

    b.   for any ordered pair of strings $S_{-m}' \ldots S_{-1}, S_1 \ldots S_n'$,

        where $m' < m$ and $n' <_{-} n$ either at most one of the 3 relations

        $\lessdot \doteq \bullet >$ holds or otherwise b. is satisfied for the pair

        $S_{-(m'+1)} \cdot \cdot \, \bullet \quad \cdots S_1 \ldots S_{n'+1} \cdot$

A precedence syntax of order (1,1) is called a simple precedence syntax. With the help of the sets of leftmost and rightmost strings, the defini- tions of the precedence relations can be reformulated analogously to their counterparts in section 2b, subject to the condition that there exists no rule U +A .

    a. $x \doteq y \leftrightarrow \exists \varphi(\varphi: U \to u S_{-1} S_1 v)$

$$\wedge (u' S_{-m} \ldots S_{-2} = u \vee S_{-m} \ldots S_{-2} \in \mathcal{R}^{(m-1)}(u))$$

$$\wedge (S_2 \ldots S_n v' = v \vee S_2 \ldots S_n \in \mathcal{L}^{(n-1)}(v))$$

    b. $x \lessdot y \leftrightarrow \exists \varphi(\varphi: u \to u S_{-1} U_1 v)$

$$\wedge (u' S_{-m} \ldots S_{-2} = u \vee S_{-m} \ldots S_{-2} \in \mathcal{R}^{(m-1)}(u))$$

$$\wedge (S_1 \ldots S_n \in \mathcal{L}^{(n)}(U_1 v))$$

    c. $x \gtrdot y \leftrightarrow \exists \varphi(\varphi: u \to u U_1 S_1 v)$

$$\wedge (S_{-m} \cdot \cdot \, \bullet \, \cdot \bigcirc \Box \in \mathcal{R}^{(m)}(u U_1) \,\,)$$

$$\wedge (S_1 \ldots S_n v' = v \vee S_2 \ldots S_n \in \mathcal{L}^{(n-1)}(v))$$

    or $\exists \varphi(\varphi: u \to u U_1 U_2 v)$

$$\wedge (S_{-m} \cdot \ldots S_{-1} \in \mathcal{R}^{(m)}(u U_1) \wedge (S_1 \ldots {}_n S \in \mathcal{L}^{(n)}(U_2 v))$$

$\mathcal{L}^{(n)}(s)$ and $\mathcal{R}^{(n)}(s)$ are then defined as follows:

1. $z = Z_1 \ldots Z_n \in \mathcal{L}^{(n)}(Uu) \leftrightarrow \exists k (1 \leq k \leq n) \ni$

   $((Z_1 \ldots Z_k \in \mathcal{L}^{(k)}(U)) \wedge (Z_k \ldots Z_n u' = u \vee Z_k \ldots \dot{n} z \in \mathcal{L}^{(n-k)}(u))$

1a. $z = z_1 \ldots Z_n \in \mathcal{L}^{(n)}(U) \leftrightarrow \exists k (0 \leq k \leq n) \ni$

   $(U \to Z_1 \ldots Z_k u \wedge Z_k \ldots Z_n \in \mathcal{L}^{(n-k)}(u))$

2. $z = Z_{\dot{n}} \ldots Z_1 \in \mathcal{R}^{(n)}(uU) \leftrightarrow \exists k (1 \leq k \leq n) \ni$

   $((u' Z_n \ldots Z_{k+1} = u \vee z_{\dot{n}} \ldots z_{k+1} \in \mathcal{R}^{(n-k)}(u)) \wedge (Z_k \ldots Z_1 \in \mathcal{R}^{(k)}(U))$

2a. $z = Z_n \ldots Z_1 \in \mathcal{R}^{(n)}(U) \leftrightarrow \exists k (0 \leq k \leq n) \ni$

   $(u \to u Z_k \ldots Z_1 \wedge Z_{\dot{n}} \ldots Z_{k+1} \in \mathcal{R}^{(n-k)}(u))$

These formulae indicate the method for effectively finding the sets $\mathcal{L}$ and $\mathcal{R}$ for all symbols in $\mathcal{V}$-$\mathcal{B}$. In particular, we obtain for $\mathcal{L}^{(1)}$ and $\mathcal{R}^{(1)}$ the definitions for $\mathcal{L}$ and $\mathcal{R}$ without superscript as defined in section 2b.

Although for practical purposes such as the construction of a useful programming language no precedence syntax of order greater than (2,2) -- or even (2,1) -- will be necessary, a general approach for the determination of the precedence relations of any order shall be outlined subsequently:

First it is to be determined whether a given syntax is a precedence syntax of order (1,1). If it is not, then for all pairs of symbols (S_i, Sk) between which the relationship is not unique, it has to be determined whether all relations will be unique between either $(S_j S_i, S_k)$ or $(S_i, S_k S_j)$, where S_j ranges over the entire vocabulary. According to the outcome, one obtains a precedence syntax of order (2,1), (1,2) or (2,2), or if some relations are still not unique, one has to try for even higher orders. If at some stage it is not possible to determine relations

between the strings with the appended symbol S_j ranging over the <u>entire</u>

vocabulary, then the given syntax is no precedence syntax at all.

Example:

$\mathcal{V} = \{A \ , \ B \ , \ \lambda \ , \ [\ , \ ]\}$

$\mathcal{B} = \{\lambda \ , \ [\ , \ ]\}$

$\Phi \ : \ A \to B$
$\quad \ A \to [\ B \ ]$
$\quad \ B \to \lambda$
$\quad \ B \to [\ \lambda \ ]$

The conflicting relations are $[\ \lessdot \ \lambda$, $[\ \doteq \ \lambda, \ \lambda \ \doteq \ ]$ and $\lambda \ \gtrdot \ ]$. But a

relation between $(S[\ , \ \lambda)$ or $(A, \ ]S)$ can be established for no symbol

S whatsoever, and between $([\ , \ \lambda S_1)$ and $(S_2 \lambda \ , \ ])$ only for $S_1 =]$

and $S_2 = [$. Thus this is no precedence syntax.

Clearly there exist two different parses for the string [A],

namely

$$[\ \lambda \ ] \qquad \text{and} \qquad [\ \lambda \ ]$$

$$B \qquad \qquad B$$

$$A \qquad \qquad A$$

_____
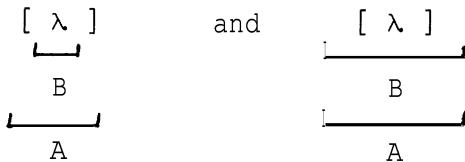
The underlying phrase structure systems in section III.3 and chapter

IV will be simple precedence phrase structure systems.

C. <u>An Example</u>

A simple phrase structure programming language shall serve as an

illustration of the presented concepts. This language contains the fol-

lowing constructs which are well-known from ALGOL 60: Variables, arithmetic

expressions, assignment statements, declarations and the block structure.

The meaning of the language is explained in terms of an array of variables,

called the 'value stack', which has to be understood as being associated
with the array $\underline{S}$ which is instrumental in the parsing algorithm. The
variable $\underline{V}_i$ represents the 'value' associated with the symbol $\underline{S}_i$.
E.g., the interpretation rule ψ_{11} corresponding to the syntactic rule
ϕ_{11} determines the value of the resulting symbol expr- as the sum of
the values of the symbols expr- and term belonging to the string to be
reduced.

ϕ_{11} : expr- $\rightarrow$ expr- + term

ψ_{11} : $\underline{v}_j \leftarrow \underline{V}_j + \underline{V}_i$, $\quad [\underline{V}(\underline{\text{expr-}}) \leftarrow V(\underline{\text{expr-}}) + V(\underline{\text{term}})]$

Note that the string to be reduced has been denoted by $\underline{S}_j \cdots \underline{S}_i$ in the
parsing algorithm of section III.2a. Instead of thus making explicit
reference to a particular parsing algorithm, $\underline{V}_i \cdots \underline{V}_i$, the values of
the symbols $\underline{S}_i \cdots \underline{S}_j$, can be denoted explicitly, i.e. instead of $\underline{V}_i$
and $\underline{V}_j$ in ψ_{11} one might write $\underline{V}(\text{term})$ and $\underline{V}(\underline{\text{expr-}})$ respectively.
For the sake of brevity, the subscripts i and j have been preferred
here.

A second set of variables is called the 'name stack' . It serves to
represent a second value of certain symbols, which can be considered as
a 'name' . The symbol decl is actually the only symbol with two values;
it represents a variable of the program in execution which has a name
(namely its associated identifier) and a value (namely the value last
assigned to it by the program). The syntax of the language is such that
the symbol decl remains in the parse-stack S as long as the declaration
is valid, i.e. until the block to which the declaration belongs is closed.
This is achieved by defining the sequence of declarations in the head
of a block by the right - recursive syntactic rule ϕ_4 . The

36

parse of a sequence of declarations illustrates that the declarations can

only be involved in a reduction together with a <u>body-</u> symbol after a

symbol <u>body-</u> has originated through some other syntactic reduction.

This, in turn, is only possibly when the symbol <u>end</u> is encountered. The

<u>end</u> symbol then initiates a whole sequence of reductions which result in

the collapsing of the part of the stack which represented the closing

block. On the other hand, the sequence of statements which constitutes

the imperative part of a block, is defined by the left-recursive syntactic

formula φ_6 . Thus a statement reduces with a preceding statement-list

into a statement-list immediately, because there is no need to retain

information about the executed statement in the value-stack.

This is a typical example where the syntax is engaged in the defini-

tion of not only the structure but also the meaning of a language. The

consequence is that in constructing a syntax one has to be fully aware of

the meaning of a constituent of the language and its interaction with

other constituents. Many other such examples will be found in chapter IV

of this article. It is, however, not possible to ennumerate and discuss

every particular consideration which had to be made during the construction

of the language. 'Only a detailed study and analysis of the language can

usually reveal the reasons for the many decisions which were taken in

its design.

Subsequently the formal definition of the simple phrase structure

language is given:

$$\mathcal{L}_p = (\mathcal{V}, \Phi, \mathcal{B}, \underline{program}, @, \&)$$

$$\mathcal{V} - \mathcal{B} = \{\underline{program} \mid \underline{block} \mid \underline{body} \mid \underline{body-} \mid \underline{decl} \mid \underline{statment}$$
$$\underline{statlist} \mid \underline{expr} \mid \underline{exp:r-} \mid \underline{term} \mid \underline{term-}$$
$$\underline{factor} \mid \underline{var} \mid \underline{number} \mid \underline{digit} \mid \bot\}$$

$$\mathcal{B} = \{\lambda \mid \underline{begin} \mid \underline{end} \mid ; \mid , \mid \leftarrow \mid + \mid - \mid \times \mid / \mid (\mid) \mid$$
$$0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid \underline{new} \mid \bot\}$$

$$\mathcal{E} = \{\underline{S}, \underline{V}, \underline{W}, \mathrm{i}\}$$

37

Φ : φ_1 :   program $\rightarrow \perp$ block $\perp$

$\quad\quad$ φ_2 :   block $\rightarrow$ begin body end

$\quad\quad$ φ_3 :   body $\rightarrow$ body-

$\quad\quad$ φ_4 :   body- $\rightarrow$ decl ; body-

$\quad\quad$ φ_5 :   body- $\rightarrow$ statlist

$\quad\quad$ φ_6 :   statlist $\rightarrow$ statlist , statment

$\quad\quad$ φ_7 :   statlist $\rightarrow$ statment

$\quad\quad$ φ_8 :   statment $\rightarrow$ var $\leftarrow$ expr

$\quad\quad$ φ_9 :   statment $\rightarrow$ block

$\quad\quad$ φ_{10} :   expr $\rightarrow$ expr-

$\quad\quad$ φ_{11} :   expr- $\rightarrow$ expr- + term

$\quad\quad$ φ_{12} :   expr- $\rightarrow$ expr- - term

$\quad\quad$ φ_{13} :   expr- $\rightarrow$ - term

$\quad\quad$ φ_{14} :   expr- $\rightarrow$ term

$\quad\quad$ φ_{15} :   term $\rightarrow$ term-

$\quad\quad$ φ_{16} :   term- $\rightarrow$ term- $\times$ factor

$\quad\quad$ φ_{17} :   term- $\rightarrow$ term- / factor

$\quad\quad$ φ_{18} :   term- $\rightarrow$ factor

$\quad\quad$ φ_{19} :   factor $\rightarrow$ var

$\quad\quad$ φ_{20} :   factor $\rightarrow$ (expr)

$\quad\quad$ φ_{21} :   factor $\rightarrow$ number

$\quad\quad$ φ_{22} :   var $\rightarrow \lambda$

$\quad\quad$ φ_{23} :   number $\rightarrow$ digit

$\quad\quad$ φ_{24} :   number $\rightarrow$ number digit

$\quad\quad$ φ_{25} :   decl $\rightarrow$ new λ

$\quad\quad$ φ_{26} :   digit $\rightarrow$ 0

$\quad\quad$ φ_{27} :   digit $\rightarrow$ 1
$\quad\quad$ . . . .

$\quad\quad$ φ_{35} :   digit   9

Ψ : ψ_1 : $\wedge$ (empty)

$\quad\quad$ ψ_2 : $\wedge$

$\quad\quad$ ψ_3 : $\wedge$

$\quad\quad$ ψ_4 : $\underline{w}_j \leftarrow \Omega$

$\quad\quad$ ψ_5 : $\wedge$

$\quad\quad$ ψ_6 : $\wedge$

$\quad\quad$ ψ_7 : $\wedge$

$\quad\quad$ ψ_8 : $\underline{V}_{\underline{V}_j} \leftarrow \underline{V}_i$

$\quad\quad$ ψ_9 : $\wedge$

$\quad\quad$ ψ_{10} : $\wedge$

$\quad\quad$ ψ_{11} : $\underline{V}_j \leftarrow \underline{V}_j + \underline{V}_i$

$\quad\quad$ ψ_{12} : $\underline{V}_j \leftarrow \underline{V}_j - \underline{V}_i$

$\quad\quad$ ψ_{13} : $\underline{V}_j \leftarrow - \underline{V}_i$

$\quad\quad$ ψ_{14} : $\wedge$

$\quad\quad$ ψ_{15} : $\wedge$

$\quad\quad$ ψ_{16} : $\underline{V}_j \leftarrow \underline{V}_j \times \underline{V}_i$

$\quad\quad$ ψ_{17} : $\underline{V}_j \leftarrow \underline{V}_j / \underline{V}_i$

$\quad\quad$ ψ_{18} : $\wedge$

$\quad\quad$ ψ_{19} : $\underline{V}_j \leftarrow \underline{V}_{\underline{V}_j}$

$\quad\quad$ ψ_{20} : $\underline{V}_j \leftarrow \underline{V}_{j+1}$

$\quad\quad$ ψ_{21} : $\wedge$

$\quad\quad$ ψ_{22} : $\begin{aligned} &t \leftarrow j \\ &t \leftarrow t - 1 \\ &\underline{if}\ t = 0\ \underline{then} \\ &\underline{if}\ W_t \neq S_j\ \underline{then} \\ &\underline{V}_j \leftarrow t \end{aligned}$ $\longrightarrow$ ERROR

$\quad\quad$ $\$23$: $\wedge$

$\quad\quad$ ψ_{24} : $\begin{aligned} \underline{V}_j &\leftarrow \underline{V}_j \times 10 \\ \underline{V}_j &\leftarrow \underline{V}_j + \underline{V}_i \end{aligned}$

$\quad\quad$ ψ_{25} : $\begin{aligned} W_j &\leftarrow S_i \\ \underline{V}_j &\leftarrow \Omega_i \end{aligned}$

$\quad\quad$ ψ_{26} : $\underline{V}_j \leftarrow 0$

$\quad\quad$ ψ_{27} : $\underline{V}_j \leftarrow 1$
$\quad\quad$ . . . .

$\quad\quad$ ψ_{35} : $V_j \leftarrow 9$

38

Notes:

1. The branch in rule ψ_{22} labelled with ERROR is an example for the indication of a 'semantic error' in $\mathcal{A}_p$. By 'semantic error' is in general meant a reaction of an interpretation rule which is not explicitly defined. In the example of ψ_{22} the labelled branch is followed when no identifier equal to $\underline{S}_i$ is found in the W stack, i.e. when an 'undeclared' identifier is encountered.

2. The basic symbol λ in $\mathcal{V}$ is here meant to act as a representative of the class of all identifiers. Nothing will be said about the representation of identifiers.

On the subsequent pages follow the sets of leftmost and rightmost symbols $\mathcal{L}$ and $\mathcal{R}$, the matrix $\underline{M}$ of precedence relations, and the precedence functions f and g, all of which were determined by the syntax-processor program listed in Appendix I.

BLOCK	BEGIN								
BODY	BODY-	DECL	NEW	STATLIST	STATMENT	VAR	IDENT	BLOCK	BEGIN
BODY-	DECL	NEW	STATLIST	STATYENT	VA9	IDENT	BLOCK	BEGIN	BEGIN
DECL	NEW								
STATLIST	STATLIST	STATMENT	VAR	IDENT	BLOCK	BEGIN			
STATYENT	VAR	IDENT	BLDCK	BEGIN					
VAR	IDENT								
EXPR	EXPR-	-	TERM	TERM-	FACTOR	VAR	IDENT	(	DIGIT	0
	1	2	3	4	5	6	7	8	9	NUMBER
EXPR-	EXPR-	-	TERM	TERM-	FACTOR	VAR	IDENT	(	DIGIT	0
	1	2	3	4	5	6	7	9	9	YUMBER
TERM	TERM-	FACTOR	VAR	IDENT	(	DIGIT	0	1	2	3
	4	5	6		8	9	NUMBER			
TERM-	TERM-	FACTOR	JAR	IDENT	(	DIGIT	0	1	2	3
	4	5	6		6	9	YUYBER			
FACTOR	VAR	IDENT	(	DIGIT	0	1	2	3	4	5
	6	7	8	9	NUMBER					
NUMBER	DIGIT	0	1	2	3	4	S	6	7	9
	9	NUMBER								
DIGIT	0	1	2	3	4	5	6	7	8	9

BLOCK	END								
BODY	BODY-	STATLIST	STATMENT	EXPR	EXPR-	TERM	TERM-	FACTOR	VAR	IDENT
	)	NUMBER	DIGIT	0	1	2	3	4	5	6
	7	8	9	RLOCK	END					
BODY-	BODY-	STATLIST	STATMENT	EXPR	EXPR-	TERM	TERM-	FACTOR	VAR	IOENT
	)	NUMBER	DIGIT	0	1	2	3	4	5	6
	7	8	9	BLOCK	END					
DCCL	IOENT								
STATLIST	STATYENT	EXPR	EXPR-	TERM	TERM-	FACTOR	VAR	IDENT	)	NUMBER
	DIGIT	0	1	2	3	4	5	6	7	8
	9	BLOCK	END							
STATYENT	EXPR	EXPR-	TERM	TERM-	FACTOR	VAR	IDENT	)	NUMBER	DIGIT
	0	1	2	3	4	5	6	7	B	9
	BLOCK	END								
VAR	IDENT								
EXPR	EXPR-	TERM	TERM-	FACTOR	VAR	IDENT	)	NUMBER	DIGIT	0
	1	2	3	4	5	6	7	8	9	
EXPR-	TERM	TERM'	FACTOR	VAR	IDENT	)	NUMBER	DIGIT	0	1
	2	3	4	5	6	7	8	9		
TERM	TERM-	FACTJR	JAR	IDENT	)	NUMBER	DIGIT	0	1	2
	3	4	5	6	7	8	9			
TERM-	FACTOR	VAR	IDENT	)	NUMBER	DIGIT	0	1	2	3
	4	5	6	7	8	9				
FACTOR	VAR	IDENT	)	NUMBER	DIGIT	0	1	2	3	4
	5	6	7	8	9					
NUMBER	DIGIT	0	1	2	3	4	5	6	7	8
	9									
DIGIT	0	1	2	3	4	5	6	7	8	9

Precedence matrix (rows = left symbol, columns = right symbol). Symbols: v = ⋖, ^ = ⋗, " = ≐

Row and column index legend:

#	Symbol
1	BLOCK
2	BODY
3	BODY-
4	DECL
5	STATLIST
6	STATMENT
7	VAR
8	EXPR
9	EXPR-
10	TERM
11	TERM-
12	FACTOR
13	NUMBER
14	DIGIT
15	IDENT
16	BEGIN
17	END
18	;
19	,
20	←
21	+
22	-
23	x
24	/
25	(
26	)
27	0
28	1
29	2
30	3
31	4
32	5
33	6
34	7
35	8
36	9
37	NEW
38	⊢

41

NO.	SYMBOL	F	G
001	BLUCK	003	004
002	BODY	001	001
003	BODY-	002	002
004	DECL	001	003
005	STATLIST	002	003
006	STATMENT	003	003
007	VAR	006	004
008	EXPR	003	001
009	EXPR-	004	002
010	TERM	005	002
011	TERM-	005	003
012	FACTOR	006	003
013	NUMBER	006	004
014	DIGIT	008	006
015	IDENT	007	004
016	BEGIN	001	005
017	END	004	001
018	;	002	001
019	,	003	002
020	←	001	006
021	+	002	004
022	-	002	004
023	×	003	005
024	/	003	005
025	(	001	004
026	)	006	003
027	0	008	007
028	1	008	007
029	2	008	007
030	3	008	007
031	4	008	007
032	5	008	007
033	6	008	007
034	7	008	007
035	8	008	007
036	9	008	007
037	NEW	004	003
038	⊥	004	003

IV. EULER: An Extension and Generalization of ALGOL 60

In this chapter the algorithmic language EULER is described first informally and then formally by its syntax and semantics. An attempt has been made to generalize and extend some of the concepts of ALGOL thus creating a language which is simpler yet more flexible than ALGOL 60. A second objective in developing this language was to show that a useful programming language which can be processed with reasonable efficiency can be defined in rigorous formality.

A. An Informal Description of EULER:

1. Variables and Constants

In ALGOL the following kinds of quantities are distinguished: simple variables, arrays, labels, switches and procedures. Some of these quantities 'possess values' and these values can be of certain types, integer, real and Boolean. These quantities are declared and named by identifiers in the head of blocks. Since these declarations fix some of the properties of the quantities involved, ALGOL is rather restrictive with respect to dynamic changes. The variables are the most flexible quantities, because values can be assigned dynamically to them. But the type of these values always remains the same. The other quantities are even less flexible. An array identifier will always designate a quantity with a fixed dimension, fixed subscript bounds and a fixed type of all elements. A procedure identifier will always designate a fixed procedure body, with a fixed number of parameters with fixed type specification (when given) and with fixed decision on whether the parameters are to be called by name or by value. A switch identifier always designates a list with a fixed number of fixed elements. We may call arrays, procedures, and switches 'semistatic',

43

because some of their properties may be fixed by their declarations.

In order to lift these restrictions, EULER employs a general type concept. Arrays, procedures, and switches are not quantities which are declared and named by identifiers*, i.e. they are not as in ALGOL quantities which are on the same level as variables. In EULER these quantities are on the level of numeric and Boolean constants. EULER therefore introduces besides the

    number   and
    logical constant

the following additional types of constants:

    reference,
    label,
    symbol
    list (array),
    procedure,
    undefined.

These constants can be assigned to variables, which assume the same form as in ALGOL, but for which no fixed types are specified. This dynamic principle of type handling requires of course that each operator has to make a type test at execution time to insure that the operands involved are appropriate.

The generality goes one step further: A procedure when executed can produce a value of any type (and can of course also act by side effects), and this type can vary from one call of this procedure to the next. The elements of a list can have values of any type and the type can be different from element to element within the same list. If the list elements are labels then we have a switch, if the elements are procedures then we have a procedure list, a construct which is not available in ALGOL 60 at all. If the elements of a list are lists themselves then we have a general tree structure.

---

* identifiers are defined in EULER exactly as in ALGOL 60.

44

EULER provides general type-test operators and some type conversion operators.

   a) <u>Numbers and Logical Constants</u>

      Numbers in EULER are essentially defined like unsigned numbers in ALGOL 60.

      The logical constants are <u>true</u> and <u>false.</u>

   b) <u>References</u>

      A reference to a variable in EULER is a value of type Reference. It designates the identity of this particular variable rather than the value assigned to it. We can form a reference by applying the operator @ to a variable:

                        @<variable>

The inverse of the reference operator is the evaluation operator (.). If a certain variable x has been assigned the reference to a variable y, then

                           x.

represents thevariable y.   Therefore the form

                        <variable>.

is also considered to be a variable.

   c) <u>Labels</u>

      A label is like in ALGOL a designation of an entry point into a statement sequence. It is a 'Program Reference' . A label is symbolically represented by an identifier. In contrast to ALGOL 60 each label has to be declared in the head of the block where it is defined. In the paragraph on declarations it is explained why this is so.

d) **Symbols**

A **symbol** (or character) in **EULER** is an éntity denoted in a dis-
tinguishable manner as a literal **symbol.** A list of symbols is
called a string.

e) Lists

Lists in EULER take the place of arrays in ALGOL. But they
are more general than arrays in ALGOL in several respects.   L i s t s
can be assigned to variables, and are not restricted to a rectangular
format; they can display a general tree structure. Furthermore,
the structure of lists can be changed dynamically by list operators.

Basically a list is a linear array of a number of elements
(possibly zero). A list element is a variable: to it can be assigned
a constant of any type (in particular, it can itself be a list), and
its identity can be specified by a reference.'

A list can be written explicitly as

(<expression> , <expression> , ...)

The expressions are evaluated in sequence and the results are the
elements of the created list.

A second way to specify a list literally is by means of the list
operator list

list <expression>

where the expression has to deliver a value of type Number, and the
result is a list with as many elements (initialized to Ω) as spe-
cified by the expression.

The elements of a list are numbered with the natural numbers
beginning with 1. A list element can be referenced by subscripting

46

a variable (or a list element) to which a list is assigned. If the subscript is not an integer then its value rounded to the nearest integer is used for subscripting. An attempt to subscript with i, where i<0 or i > length of the list, results in an error indication. An example for specifying a list structure is

$$(1,2,(3,(4,5),6,())) .$$

This is a list with three elements, the first two elements being numbers, the third element being a list itself. This **sublist** has four elements, a number, another **sublist**, again a number and last another **sublist** with 0 elements. If this list would have been assigned to the variable a, then a[2] would be the number 2, a[3][2] would be the list (4,5) .

In order to manipulate lists, list operators are introduced into EULER. There are a type-test operator (isli), an operator to determine the current number of elements (length), a concatenation operator (&), and an operator to delete the first element of a list (tail). Here are some examples for the use of these operators: (Assuming the list given above assigned to a)

```
isli a[2]               gives a value false
length a[3][4]          gives a value 0
(2,3) & a[3][2]         gives the list  (2,3,4,5)
(a[2]) & tail tail a[3] gives the list (2,6,())
```

From the formal description of EULER it can be seen what rules have to be observed in applying list operators, and in what sequence these operators are executed when they appear together in an expression (like in the last example).

Only a minimal set of list operators is provided in EULER. This set can, however, easily be expanded. The introduction of list

manipulation facilities into EULER makes it possible to express with
this language certain problems of processing symbolic expressions
which can not be handled in ALGOL but required special list processing
languages like LISP or IPL.

f) Procedures

Similar to ALGOL, a procedure is an expression which is defined
once (possibly using formal parameters), and which can be evaluated
at various places in the program (after substituting actual para-
meters). The notion of a procedure in EULER is, however, in several
respects more general than in ALGOL. A procedure, i.e. the text
representing it, is considered a constant, and can therefore be
assigned to a variable. An evaluation of this variable'effects an
evaluation of this procedure, which always results in a value. In
this respect every EULER procedure acts like a type-procedure in ALGOL.
The number and type of parameters specified may vary from one call of
a procedure to the next call of this same procedure.

Formally parameters are always called 'by value' . However,
since an actual parameter can again be a procedure, the equivalent
of a "call by name' in ALGOL can be accomplished. Furthermore an
actual parameter being a reference establishes a third kind of call:
"call by reference'. It must be-noted that the type of the call of
a parameter is determined on the calling side. For example, assuming
i = 1 and a[i] = 2,

    p(a[i])   is a call by value,
    p(‘a[i]’)   is a call by procedure (name),
    p(@ a[i])   is a call by reference.

48

In the first case the value of the parameter is 2,   in the second

case it is a[i], in the third case it is the reference to a[1] .

A procedure is written as

$$' \text{<expression>} ' \quad \text{or}$$

$$' \delta;\delta;...;\delta; \text{<expression>}'$$

where δ represents a formal declaration. The evaluation of a

procedure yields the expression enclosed in the quote marks.

A formal declaration is written as

formal <identifier> .

The scope of a formal variable is the procedure and the value assigned

to it is the value of the actual parameter if there exists one, Ω

otherwise. When a formal variable is used in the body of the proce-

dure, an evaluation of it is implied. For instance in

$$p \leftarrow ' \underline{formal} \ x; \ x \leftarrow 5' ;...;p(@ \ a);$$

the reference to a is assigned to the formal variable x, and the

implied evaluation of x causes the number 5 to be assigned to

the variable a (and not to the formal variable x). As a conse-

quence, the call p(1) would imply that an assignment should be made

to the constant 1. This is not allowed and will result in an error

indication.

g) The **Value** '**Undefined**'

The constant Ω means 'undefined? Variables are automatically

initialized to this value by declarations. **Also,a** formal parameter

is assigned this value when a procedure is called and no corresponding

actual parameter is specified in the calling sequence.

2. Expressions

In ALGOL an expression is a rule for obtaining a value by applying certain operators to certain operands, which have themselves values. A statement in ALGOL is the basic unit to prescribe actions. In EULER these two entities are combined and called 'expression', while the term 'statement' is reserved for an expression which is possibly labelled. An expression in EULER, with the exception of a **goto-expression,** produces a value by applying certain operators to certain operands, and at the same time may cause side effects. The basic operands which enter into expressions are constants of the various types as presented in paragraph 1, variables and list elements, values read in from input devices, values delivered by the execution of procedures and values of expressions enclosed in brackets. Operators are in general defined selectively to operate on operands of a certain type and producing values of a certain type. Since the type of a value assigned to a variable can vary, **type**-tests have to be made by the operators at execution time. If a type test is unsuccessful, an error indication is given. Expressions are generally executed from left to right unless the hierarchy between operators demands execution in a different sequence. The hierarchy is implicitly given by the syntax.

Operators with the highest precedence are the following type test operators:

```
isb  <variable>   (is logical?)
isn  <variable>   (is number?)
isr  <variable>   (is reference?)
isl      .        (is label?)
isy      .        (is symbol?)
isli     .        (is list?)
isp      .        (is procedure?)
isu  <variable>   (is undefined?)
```

These operators, when applied to a variable, yield _true_ or _false_,
depending upon the type of the value currently assigned to the variable.
At the same level are the numeric unary operators: _abs_ (forming the
absolute value of an operand of type Number), _integer_ (rounding an operand
of type Number to its nearest integer), the list reservation operator
list, the length operator _length_ (yielding the number of elements in a
list), the _tail_ operator, and type conversion operators like _real_, which
converts a logical value into a number, _logical_ which converts a number
into a logical value, conversion operators from numbers to symbols and
from symbols to numbers, etc.

The next lower precedence levels contain in this sequence: **Exponen**-
tiation operator, multiplication operators ($\times, /, \div$, _mod_), addition op-
erators (+, -), extremal operators (max, min). Operands and results
are of **type** Number.

The next lower precedence levels contain the relational and logical
operators in this sequence: relational operators ($=, \neq, <, \leq, >, \geq$),
negation operator $\neg$, conjunction operator $\wedge$, disjunction operator $\vee$.
The relational operators require that their operands are of type Number
and they form a logical value. The operators $\wedge$ and $\vee$ are executed
differently from their ALGOL counterparts: If the result is already
determined by the value of the first operand, then the second operand
is not evaluated at all. Thus, _false_ $\wedge$ x $\rightarrow$ _false_, true $\vee$ x $\rightarrow$ _true_ for
all x.

The next lower precedence level contains the concatenation operator
& .

Operators of the lowest level are the sequential operators _goto_,

If, then, and else, the assignment operator ←, the output operator
out and the bracketing symbols begin and end. According to their
occurence we distinguish between the following types of expressions:
goto-expression, assignment expression, output expression, conditional
expression, and block. As it was already mentioned, all expressions
except the goto-expression produce a value, while in addition they may
or may not cause a side effect.

The go-to-expression is of the form

<p align="center">go <expression></p>

If the value of the expression following the goto-operator is of the type
Label, then control is transferred to the point in the program which this
label represents. If this expression produces a value of a different type,
then an error indication is given.

The assignment expression assigns a value to a variable. It is of
the form

<p align="center"><variable> ← <expression></p>

In contrast to ALGOL an assignment expression produces a value, namely
the value of the expression following the assignment operator, This
general nature of the EULER assignment operator allows assignment of
intermediate results of an expression. For example:

<p align="center">a ← b + [c ← d + e]</p>

would compute d + e, assign this result to c, and then add b, and
assign the total to a.

The output expression is of the form

<p align="center">out <expression></p>

The value of the expression following the output operator is transmitted

<p align="center">52</p>

to an output medium . The value of the output expression is the value of the expression following the output operator.

A conditional expression is of the form

<div style="text-align:center">

.if <expression> <u>then</u> **<expression>** <u>else</u> <expression>

</div>

The meaning is the same as in ALGOL.

The construct

<div style="text-align:center">

<u>if</u> <expression> <u>then</u> **<expression>**

</div>

is not allowed in EULER, because this expression would not produce a **value,**if the value of the first expression is <u>false</u>.

An expression can also be a block.

3. Statements and Blocks

A statement in EULER is an expression which may be preceded by one or more label definition'(s). If a statement is followed by another statement, then the two statements are separated by a semicolon. A semi-colon discards the value produced by the previous statement. Since a goto-expression leads into the evaluation of a statement without encoun-tering a semicolon, the **goto** operator also has to discard the value of the statement in which it appears.

A block in EULER is like in ALGOL a device to delineate the scope of identifiers used for variables and labels, and to group statements into statement sequences. A block is of the form

<div style="text-align:center">

<u>begin</u> $\sigma;\sigma;\dots;\sigma$ <u>end</u>   or

<u>begin</u> $\delta;\delta;\dots;\delta;\sigma;\sigma;\dots;\sigma$ <u>end</u>

</div>

where σ represents a statement and δ represents a declaration. The last statement of a block is not followed by a semicolon, therefore its value becomes the value of the block.

<div style="text-align:center">53</div>

Since procedures, labels, and references in EULER are quantities which can be dynamically assigned to variables, there is a problem which is unknown to ALGOL: These quantities can be assigned to variables which in turn can be evaluated in places where these quantities or parts of them are undefined.

Situations like this are defined as semantic errors, i.e. the language definition is such that occurrences of these situations are detected.

4. Declarations

There are two types of declarations in EULER, variable-declarations and label-declarations:

    new <identifier>   and
    label <identifier>

A variable declaration defines a variable for this block and all inner blocks, to be referenced by this identifier as long as this same identifier is not used to redeclare a variable or a label in an inner block. A variable declaration also assigns the initial value Ω to the variable. As discussed in paragraph 1, no fixed type is associated with a variable.

A label declaration serves a different purpose. It is not a definition like the variable declaration; it is only an announcement that there is going to be a definition of a label in this block of the form

                    <identifier> :

prefixing a statement.

Although the label declaration is dispensable it is introduced into EULER to make it easier to handle forward references. A situation like

        begin...L:...begin...goto L;...L:...end;..end

makes it necessary to detect that the identifier L following the **goto** operator is supposed to designate the label defined in the inner block. Without label'declarations it is impossible to decide, whether an identifier (not declared in the same block) refers to a variable declared in an outer block, or to a label to be defined later in this block, unless the whole block is scanned. With a label declaration every identifier is known upon encounter.

B. The Formal Definition of EULER

EULER was to be a language which can be concisely defined in such a way that the language is guaranteed to be unambiguous, and that from the language definition a processing system can be derived mechanically, with the additional requirement that this processing system should run with reasonable efficiency. A method to perform this transformation mechanically, and to accomplish parsing efficiently, has been developed and is given in Chapter III for languages which are simple precedence phrase structure languages. Therefore, it appeared to be highly desirable to define EULER as a simple precedence language with precedence functions. It was possible to do this and still include in EULER the main features of ALGOL and generalize and extend ALGOL as described.

The definition of EULER is given in two 'steps' to insure that the language definition itself forms a reasonably efficient processing system for EULER texts. The definition of the compiling system consists of the parsing algorithm, given in paragraph III.B.1., a set of syntactic rules, and a set of corresponding interpretation rules by which an EULER text is transformed into a polish string. The definition of the executing system consists of a basic interpreting mechanism with a rule to interpret each symbol in the polish string. Both descriptions use the basic notation of chapter II. If the definition of EULER would have been given in one step like the definition of the example in chapter III C, it would have been necessary to transform it into a two phase system in order to obtain an efficient processing system. Furthermore, a one phase definition requires   the introduction of certain concepts (e.g. a passive mode, where a text is only parsed but not evaluated) which are without consequence for

practical systems, because they take on an entirely different appearance when transformed into a two phase system.

The form of the syntactic definition of EULER is influenced by the requirement that EULER be an unambiguous simple precedence phrase structure language. This involves that:

a) there must be exactly one interpretation rule (possible empty) for each syntactic rule,

b) the parsing algorithm has to find reducible substrings in exactly the same sequence in which the corresponding interpretation rules have to be obeyed,

c) extra syntactic classes (with empty interpretation rules) have to be introduced to insure that at most one precedence relation holds between any two symbols,

d) no two syntactic rules can have the same right part.

For an illustration of the requirements a) and b) consider the syntactic definition of an arithmetic expression in ALGOL 60:

```
<arithmetic expression> :: = <simple arithmetic expression> |
        <if clause> <simple arithmetic expression> else
                    <arithmetic expression>
```

If the text

                    if b then a + c else d + e

is parsed, then d + e is reduced to <arithmetic expression> and accordingly evaluated, before it has been taken into account that the preceding <if clause> may prevent d + e to be evaluated at all. In this example, the syntax of ALGOL 60 fails to reflect the sequence of evaluation properly, as it does e.g. in the formulations of simple expressions. To correct this default, the corresponding syntactic definitions in EULER are as follows: (BNF is adopted here to obviate the analogies)

```
<expresssion> ::= <if clause> <true part> <expression>
<if clause> ::= if <expression> then'
<true part> ::= <expression> else
```

In the example above, the operator else will be recognized as occuring in

<true part> before the expression d + e is parsed. Through the inter-

pretation rule for <true part> the necessary code can be generated.

A similar situation holds for the ALGOL definition

<basic statement> ::= <label> : <basic statement>

The colon, denoting the definition of a label, is included in a reduction

only after <basic statement> was parsed and evaluated. In EULER the

corresponding definitions read:

```
<statement> ::= <label definition> <statement>
<label definition> ::= <identifier> :
```

Thus the parsing algorithm detects the label definition before parsing the

statement.

As a third example, we give the EULER definition of <disjunction>

```
<disjunction> ::= <disjunction head> <disjunction>
<disjunction head> ::= <conjunction> V
```

Thus, V is included in a syntactic reduction, before <disjunction> is

parsed and evaluated; code can be generated which allows conditional skip-

ping of the following part of program corresponding t&disjunction>.

The corresponding ALGOL syntax

<Boolean term> ::=.<Boolean term> V <Boolean factor>

reflects the fact that both <Boolean term> and <Boolean factor> are

to be evaluated before the logical operation is performed. This inter-

pretation of the logical operators ∧ and V was deliberately discarded

as being undesirable.

According to requirement c) the language definition of EULER

contains certain auxiliary nonbasic symbols like

<variable-> , <integer-> etc. to insure that EULER is a simple
precedence language. Without these nonbasic symbols the reducible sub-
strings in a sentence are not disjoint, as the following example taken
from ALGOL shows:

$$
\underbrace{x \quad \times}_{\text{<term>}} \quad \underbrace{y \quad \uparrow \quad z}_{\text{<factor>}} \quad , \quad x \quad \underbrace{\times \quad y}_{\text{<factor>}} \quad \uparrow \quad \underbrace{z}_{\text{<prim>}}
$$

$$
\underbrace{\hspace{5cm}}_{\text{<term>}} \qquad \underbrace{\hspace{5cm}}_{\text{<factor>}}
$$

Therefore one obtains the contradicting precedence relations $\times \doteq$ <factor>
and $\times \lessdot$ <factor> .

The requirement d) together with the precedence property is a suf-
ficient condition for the language to be unambiguous. Requirement d)
has far reaching consequences on the form of the language definition,
because it forces the syntax to be written in a sort of linear arrangement
rather than a net. Two examples will be given.

A label unlike in ALGOL can in EULER not be defined as <identifier>,
because we already have

<p align="center"><variable-> ::= <identifier></p>

This suggests that the best thing to do would be to introduce two different
forms of identifiers for the two different entities variable and label.
It was felt, however, that tradition dictates that the same kind of iden-
tifiers be used for variables and labels. It was possible to do this in
EULER although the solution might not be considered clean. In the text

<p align="center"><u>goto</u> L</p>

the identifier L is categorized by the parsing algorithm into the syn-
tactic class <variable>, but the corresponding interpretation rule ex-
amines the table of declared identifiers and discovers that this identifier

designates a label (defined or undefined at this time). Therefore, a
label is inserted into the polish string instead of a variable.

A second example for the specific arrangement of the syntax chosen
to fullfill requirement d) is the following: The concatenation operator
(&) is introduced into the syntax in the syntactic class **<catena>**,
which is defined as

$$\text{<catena>} ::= \text{<catena>} \text{ \& } \text{<primary>} \mid$$
$$\text{<disjunction>}$$

This looks as if **&** had a lower precedence than the logical and arithme-
tic operators. But this is of no consequence, since an operand of **&**
must be a quantity of type List and a <disjunction> can only be of type
List if it is a <primary>, i.e. not containing any logical or arithmetic
operators.

_ But we cannot write

$$\text{<catena>} ::= \text{<primary>} \text{ ,}$$

because this would violate requirement d). Therefore **<catena>** appears
in the syntax at a rather arbitrary place between <primary> and <expres-
sion>.

Looking at the requirements made upon the language definition and
observing the careful choices that had to be made in drawing up the
language definition in line-with these requirements, the criticism will
probably be raised, that the difficulties usually encountered in deriving
syntax directed compilers for given languages are not avoided in EULER
but merely 'sneaked' into the definition of the language itself. This
point is well taken, but we think that nobody is likely to create some-
thing as complicated as a processing system for an algorithmic language
like ALGOL without encountering some difficulties somewhere. We think

it is the merit of this method of language definition to bring these dif-
ficulties into the open, so that the basic concepts involved can be recog-
nized and systematically dealt with.   It is no longer possible to draft
an 'ad hoc syntax' and call it a programming language, because the natural
relationship between structure and meaning must be established.

Subsequently follows the formal definition of EULER.   It has been
programmed as an Extended ALGOL program for the Burroughs B5500 computer.
This program is listed in Appendix II.

Phase I (Translator)

The vocabulary $\mathcal{V}$:

The set of basic symbols $\mathcal{B}$: *

0|1|2|3|4|5|6|7|8|9|,|.|;|:|@ |new|formal|label|λ|[|]|begin|end|

(|)|'|'|goto|out|←|if|then|else|&|∨|∧|¬|=|≠|<|≤|≥|>|

min|max|+|-|×|/|÷|mod|↑|abs|length|integer|real|logical|

list|tail|in|isb|isn|isr|isl|isli|isy|isp|isu|σ|Ω|

10|÷|true|false|⊥

The set of non-basic symbols $\mathcal{V}$-$\mathcal{B}$:   -

program|block|blokhead|blokbody|labdef|stat|stat-|

expr|expr-|ifclause|truepart|catena|disj|disjhead|

conj|conj-|conjhead|negation|relation|choice|choice-|

sum|sum-|term|term-|factor|factor-|primary|procdef|

prochead|list*|reference|number|real*|

integer*|integer-|digit|logval|var|var-|vardecl|

labdecl|fordecl

The environment $\mathcal{E}_1$:

```
S       (stack used by the parsing algorithm)
V
i       (index to  S  and  V)
j       (index to  S  and  V)
P       (program produced by Phase I)
k       (index to P)
N       (list of identifiers and associated data)
n       (index to N)
m       (index to N)
bn      (block number)
on      (ordinal number)
scale   (scale factor for integers)
```

$\mathcal{E}_1$ = (S , V , i , j , P , k , N , n , m , bn , on , scale]

---
* λ and σ are representatives for identifiers and symbols respectively.

	Φ	$\Psi 1$

1: $\underline{vardecl} \rightarrow \underline{new}\ \lambda$ k ← k+1; P[k] ← ('new'); on ← on +1;
n ← n+1; N[n] ← (V[i], bn, on, 'new')

2: $\underline{fordecl} \rightarrow \underline{formal}\ \lambda$ k ← k+1; P[k] ← ('formal'); on ton +1;
n ← n+1; N[n] ← (V[i], bn, on, 'formal')

3: $\underline{labdecl} \rightarrow \underline{label}\ \lambda$ n ← n+1; N[n] ← (V[i], bn, Ω, Ω)

4: $\underline{var-} \rightarrow \lambda$

```
      t tn;  k ← k+1;
L41:  if t < 1 then goto Error:
      if N[t][1] = V[i] then goto L42;
      t ← t-1; goto L41;
L42:  if N[t][4] ≠ 'new' then goto L43 ;
      P[k] ← ('@' , N[t][3], N[t][21]); goto L46
L43:  if N[t][4] ≠ 'label' then goto L44;
      P[k] ← ('label', N[t][3], N[t][2]); goto L46
L44:  if N[t][4] ≠ 'formal' then goto L45;
      P[k] ← ('@ ', N[t][3], N[t][2]);
      k ← k+1; P[k] ← ('value');  goto L46
L45:  P[k] ← ('label', N[t][3], N[t][2]);
      N[t][3] t-k; poto L46;
L46:
```

5: $\underline{var-} \rightarrow [\underline{expr}]\underline{\quad}$ k ← k+1; P[k] ← (']')

6: $\underline{var-} \rightarrow \underline{var-}\ .$ k ← k+1; P[k] ← ('value')

7: $\underline{var}\ .\rightarrow \underline{var-}$ Λ

8: $\underline{logval} \rightarrow \underline{true}$ V[j] ← true

9: $\underline{logval} \rightarrow \underline{false}$ V[j] ← false

10: $\underline{digit}\ 4\ 0$ V[j] ← 0

....

19: $\underline{digit}\ 9$ V[j] ← 9

20: $\underline{integer-} \rightarrow \underline{digit}$ scale ← -1

21: $\underline{integer-}\ +\ \underline{integer-}\ \underline{digit}$ t ← 10 × V[j]; V[j] ← V[i] + t;
scale ← scale - 1

22: $\textbf{integer*}\ 3\ \underline{integer-}.$ Λ

23: $\underline{real*} \rightarrow \textbf{integer*}\ .\ \textbf{integer*}$ t ← 10 ↑ scale;
t ← V[i] × t;
V[j] ← V[j] + t

24: $\underline{real*}\ 3\ \textbf{integer*}$ Λ

25: number → real* ∧

26: number → real* 10 integer*   t ← 10 ↑ V[i];
V[j] ← V[j] × t

27: number → 10 ∸integer*   t ← 0.1 ↑ V[i];
V[j] ← V[j] × t

28: number → 10 integer*   V[j] ← 10 ↑ V[i]

29: number → 10 ∸ integer*   V[j] ← 0.1 ↑ V[i]

30: reference → @var   ∧

31: listhead → listhead expr ,   V[j] ← V[j] + 1

32: listhead → (V[j] ← 0

33: list* → listhead expr)   k ← k+1; P[k] ← ('')', V[j] + 1)

34: list* → listhead)   k ← k+1; P[k] ← ('')', V[j])

35: prochead → prochead fordecl ;   ∧

36: prochead → '   bn ← bn+1; on ← 0; k ← k+1;
P[k] ← ('''',Ω); V[j] tk;
n ← n+1; N[n] ← (Ω, m);
m t n

37: procdef → prochead expr '   k ← k+1; P[k] ← ('''');
P[V[j][2] ← k+1; bn tbn - 1;
n tm−1; m ← N[m][2]

38: primary → var   k ← k+1; P[k] ← ('value')

39: primary → var list*   k ← k+1; P[k] ← ('call')

40: primary → logval   k ← k+1; P[k] ← ('logval', V[j])

41: primary → number   k ← k+1; P[k] ← ('number', V[j])

42: primary → σ   k ← k+1; P[k] ← ('symbol',V[j])

43: primary → reference   A

44: primary → list*   A

45: primary → tail primary   k ← k+1; P[k] ← ('tail')

46: primary → procdef   A

47: primary → Ω   k ← k+1; P[k] ← ('Ω')

64

48: primary → [expr]   ∧

49: primary → in   k ← k+1; P[k] ← ('in")

50: primary → isn var   k ← k+1; P[k] ← ('isb')

51: primary → isn var   k ← k+1; P[k] ← ('isn')

52: primary → isr var   k ← k+1; P[k] ← ('isr')

53: primary → isl var   k ← k+1; P[k] ← ('isl')

54: primary + isli var   k ← k+1; P[k] ← ('isli')

55: primary → isy var   k ← k+1; P[k] ← ('isy')

56: primary → isp var   k ← k+1; P[k] ← ('isp')

57: primary → isn var   k ← k+1; P[k] ← ('isn')

58: primary → abs primary   k ← k+1; P[k] ← ('abs')

59: primary → length var   k ← k+1; P[k] ← ('length')

60: primary → integer primary   k ← k+1; P[k] ← ('integer')

61: primary → real primary   k ← k+1; P[k] ← ('real')

62: primary → logical primary   k ← k+1; P[k] ← ('logical')

63: primary → list primary   k ← k+1; P[k] ← ('list')

64: factor- → primary   ∧

65: factor- → factor- ↑ primary   k ← k+1; P[k] ← ('↑')

66: factor → factor-   ∧

67: term- → factor   ∧

68: term- → term- X factor   k ← k+1; P[k] ← ('X')

69: term- → term- / factor   k ← k+1; P[k] ← ('/')

70: term- → term- ÷ factor   k ← k+1; P[k] ← ('÷')

71: term- → term- mod factor   k ← k+1; P[k] ← ('mod')

72: term → term-   ∧

73: sum- → term   ∧

74: sum-   ++ term $\wedge$

75: sum-   -+-term $k \leftarrow k+1;\ P[k] \leftarrow (`\dot{-}`)$

76: sum- $\rightarrow$ sum- + term $k \leftarrow k+1;\ P[k] \leftarrow (`+`)$

77: sum- $\rightarrow$ sum- -term $k \leftarrow k+1;\ P[k] \leftarrow (`-`)$

78: sum $\rightarrow$ sum- $\wedge$

79: choice-   + sum $\wedge$

80: choice- $\rightarrow$ choice- min sum $k \leftarrow k+1;\ P[k] \leftarrow (`\underline{min}`)$

81: choice- $\rightarrow$ choice- max sum $k \leftarrow k+1;\ P[k] \leftarrow (`\underline{max}`)$

82: choice $\rightarrow$ choice- $\wedge$

83: relation $\rightarrow$ choice $\wedge$

84: relation $\rightarrow$ choice = choice $k \leftarrow k+1;\ P[k] \leftarrow (`=`)$

85: relation $\rightarrow$ choice $\neq$ choice $k \leftarrow k+1;\ P[k] \leftarrow (`\neq`)$

86: relation $\rightarrow$ choice < choice $k \leftarrow k+1;\ P[k] \leftarrow (`<`)$

87: relation $\rightarrow$ choice $\leq$ choice $k \leftarrow k+1;\ P[k] \leftarrow (`\leq`)$

88: relation $\rightarrow$ choice $\geq$ choice $k \leftarrow k+1;\ P[k] \leftarrow (`\geq`)$

89: relation $\rightarrow$ choice > choice $k \leftarrow k+1;\ P[k] \leftarrow (`>`)$

90: negation $\rightarrow$ relation $\wedge$

91: negation $\rightarrow$ ⌐ relation $k \leftarrow k+1;\ P[k] \leftarrow (`⌐`)$

92: conjhead $\rightarrow$ negation A $k \leftarrow k+1;\ P[k] \leftarrow (`A`, \Omega);\ V[j] \leftarrow k$

93: conj- $\rightarrow$ conjhead conj $P[V[j]][2] \leftarrow k+1$

94: conj- $\rightarrow$ negation $\wedge$

95: conj $\rightarrow$ conj- $\wedge$

96: disjhead $\rightarrow$ conj V $k \leftarrow k+1;\ P[k] \leftarrow (`V`, \Omega);\ V[j] \leftarrow k$

97: disj $\rightarrow$ disjhead disj $P[V[j]][2] \leftarrow k+1$

98: disj $\rightarrow$ conj $\wedge$

99: catena   3 catena & primary $k \leftarrow k+1;\ P[k] \leftarrow (`\&`)$

66

100: <u>catena</u> → <u>disj</u> $\wedge$

101: <u>truepart</u> 3 <u>expr</u> else   k ← k+1; P[k] ← (' <u>else</u> ', Ω); V[j] ← k

102: <u>ifclause</u> → if <u>expr</u> then   k ← k+1; P[k] ← (' <u>then</u> ', Ω); V[j] ← k

103: <u>expr-</u>   → '<u>Block</u> $\wedge$

104: <u>expr-</u>   → <u>ifclause</u> <u>truepart</u>
                          <u>expr-</u>   P[V[j]][2] ←V[j+1] +1; P[V[j+1]][2] ← k+1

105: <u>expr-</u>   → var ← <u>expr-</u>   k ← k+1; P[k] ← ('t')

106: <u>expr-</u>   → <u>goto</u> i m a r y   k ← k+1; P[k] ← (' <u>goto</u> ')

107: <u>expr-</u>   → <u>out</u> <u>expr-</u>   k ← k+1; P[k] ← ('<u>out</u>')

108: <u>expr-</u>   → <u>catena</u> $\wedge$

109: expr   → <u>expr-</u> $\wedge$

110: <u>stat-</u>   → labdef <u>stat-</u> $\wedge$

111: <u>stat-</u>   → ·<u>expr</u> $\wedge$

112: <u>stat</u>   → <u>stat-</u> $\wedge$

113: <u>labdef</u> → λ :   t t n ;
                              L1131: <u>if</u> t $\leq$ m <u>then goto</u> ERROR;
                                     <u>if</u> N[t][1] = V[j] <u>then goto</u> L1132;
                                     t ← t-1; <u>goto</u> L1131;
                              L1132: <u>if</u> N[t][4] $\neq$ Ω <u>then goto</u> ERROR;
                                     s ← N[t][3]; N[t][3] ← k+1;
                                     N[t][4] ← ' <u>label</u> ';
                              L 1133: <u>if</u> s = Ω <u>then goto</u> L1134;
                                     t ← P[s][2]; P[s][2] ← k+1;
                                     s ← t; <u>goto</u> L1133;
                              L1134:
114: <u>blockhead</u> → <u>begin</u>   bn ← bn+1; on ← 0; k ← k+1;
                                           P[k] ← (' <u>begin</u> ');
                                           n ← n+1; N[n] ← (Ω, m); m ← n

115: <u>blokhead</u> → <u>blokhead</u> vardecl; $\wedge$

116: <u>blokhead</u> → <u>blokhead</u> labdecl; $\wedge$

117: <u>blokbody</u> → <u>blokhead</u> $\wedge$

118: <u>blokbody</u> → <u>blokbody</u> stat;   k ← k+1; P[k] ← (' ; ')

119: <u>block</u>   → <u>blokbody</u> stat end k ← k+1; P[k] ← ('end');
                                           bn tm-1; m ← N[m][2]
120: <u>program</u> → $\perp$ <u>block</u> $\perp$ $\wedge$

67

The following is the definition of the execution code produced by Phase I.

The variables involved are:

    S   (tree structured memory stack)
    i   (stack index)
    mp (stack index, points at the last
           element of a linked list of Marks)
    P   (program)
    k   (program index of the instruction
           currently being interpreted)
    fct (counter of formal parameters)

s , t , A , B , C (variables and labels local to any interpretation

rule)

$$\mathcal{E}_2 = \{ \ S \ , \ i \ , \ mp \ , \ P \ , \ k \ , \ fct \ \}$$

The following types of quantities are introduced, which were not men-

tioned in Chapter II :

    labels   (i.e. program references)
    procedures   (i.e. procedure descriptors)

with the accompanying type-test operators isl, isp and the following

type-conversion operators :

    progref   converting the two integers pa and bn into the pro-

                gram-reference with address pa defined in the block

                with number bn.

    proc   converting three integers (block-number, Mark-index,

                program-address) into a uniquely defined **procedure-**

                descriptor,

    b l n   converting a procedure-descriptor into its block-number,

    mix   converting a procedure-descriptor or a label into the

                index of the Mark belonging to the block in which the

                procedure-descriptor or label is defined (Mark-index),

<u>adr</u>   converting a procedure-descriptor or a label into its

          program address.

Also, there exists an operator

     <u>reference</u>   converting the two integers on and bn,   into the

          reference of the variable with ordinal number on in the

          variable-list of the block with number bn.

 The detailed description of these operators depends on the particular

scheme of referencing used in an implementation, for which an example is

given in Appendix II. It should be noted, however, that a reference,

label or procedure-descriptor, may become undefined if it is assigned

to any variable which is not in its scope. Since procedures and blocks

may be activated recursively, the actual identity of a reference, label

or procedure-descriptor can only be established in Phase II, which makes

it necessary for Phase I to describe them in terms of more than one quan-

tity. The sufficient and necessary amount of information to establish

these identities is contained in the 'Marks' stored in S . Marks are

created upon entry into a block (or procedure) and deleted upon exit.

A Mark contains the following data:

     1. a block-number
     2. a link to its dynamically enclosing block
     3. a link to its statically enclosing block
     4. a list of its variables
     5: a program return address

By 'link' is meant the index of the Mark of the indicated block. ——

The following list indicates to the left the operator $P[k][l]$ currently

to be executed, and to the right the corresponding interpretation

algorithm. At the end of each rule a transfer to the Cycle routine has

to be implicitly understood. This basic fetch cycle is represented

as follows:

```
Initialize:    i ← 0; mp ← 0; k ← 0;
Cycle       :   k ← k+1;
T           :   Obey the Rule designated
                by P[k][l]; goto Cycle
```

Operators Interpretation Rules (Ψ_2)

+
```
                if ¬ isn S[i-1] then goto ERROR:
                if ¬ isn S[i] then goto ERROR;
                S[i-1] ← S[i-1] + S[i]; i ← i-1
```

-
x
/
÷
mod
}   defined analogously to +

.
```
                if ¬ isn S[i] then goto ERROR;
                S[i] ← - S[i]
```

abs
integer
logical
}   defined analogously to ·

real
```
                if ¬ isb S[i] then goto ERROR;
                S[i] ← real S[i]
```

min
```
                if ¬ isn S[i-1] then goto ERROR;
                if ¬ isn S[i] then goto ERROR;
                i ← i-1;
                if S[i] < S[i+1] then goto A;
                S[i] ← S[i+1]; A:
```

max   defined analogously to min

isn
```
                if ¬ isr S[i] then goto A;
                S[i] ← S[i]. ;
                A: S[i] ← isn S[i]
```

isb
isr
isl
isli
isy
isp
isu
}   defined analogously to isn

70

```
<               if ¬ isn S[i-1] then goto ERROR;
                if ¬ isn S[i] then goto ERROR;
                S[i-1] ← S[i-1] < S[i]; it i-1

<
≤
>
≥               defined analogously to <
=
≠


∧               if ¬ isb S[i  then goto ERROR;
                if S[i] then goto A;
                k ← P[k][2]; goto T;
                A: i - i -1


∨               if ¬ isb S[i] then goto ERROR;
                if¬ S[i] then goto A;
                k ← P[k][2]; goto T;
                A: i ← i - 1


┐               if ¬ isb S[i] then goto ERROR;
                S[i] ←┐  S[i]    ----


then            if ¬ isb S[i] then goto ERROR;
                i ← i - 1;
                if S[i+1] then goto A;
                k ← P[k][2]; poto T ;
                A:


else            k ← P[k][2]; goto T


length          if ¬ isr S[i] then goto A;
                S[i] ← S[i].;
                A: if ¬ isli S[i] then goto ERROR;
                S[i] ← length S[i]


tail            if ¬ isli S[i] then goto ERROR;
                S[i] ← tail S[i]


&               if ¬ isli S[i-1] then goto A;
                if ¬ isli S[i] then goto ERROR;
                S[i-1] ← S[i-1] & S[i]; i ← i - 1



list            A: if ¬ isn S[i] then poto ERROR;
                t ← S[i]; S[i] ← ();
                B: if t ≤ 0 then goto C;
                S[i] ← S[i] & (Ω); t ← t - 1;
                goto B; C:
```

```
number      i ← i+1; S[i] ← P[k][2]

logval      i ← i+1; S[i] ← P[k][2]

Ω           i ← i+1; S[i] ← Ω

string      i ← i+1; S[i] ← P[k][2]

label       i ← i+1; S[i] ← progref(P[k][2], P[k][3])

@           i ← i+1; S[i] ← reference(P[k][2], P[k][3])

new         S[mp][4] ← S[mp][4] & (Ω)

formal      fct ← fct+1;
            if fct < length S[mp][4] then goto A;
            S[mp][4] ← S[mp][4] & (Ω); A:

←           if ¬ isr S[i-1] then goto ERROR;
            S[i-1]. ← S[i]; S[i-1] ← S[i]
            i ← i-1;

            i ← i - 1

            if ¬ isn S[i] then goto ERROR;             (subscript)
            if S[i] < 0 then goto ERROR; i ti-1;
            if ¬ isr S[i] then goto ERROR;
            S[i] ← S[i].;
            if ¬ isli S[i] then goto ERROR;
            t ← length S[i];
            if S[i+1] > t then goto  R ;
            S[i] ← @ S[i][S[i+1]]

begin       i ← i+1;
            S[i] ← (S[mp][1]+1, mp, mp, ());           (a Mark)
            mp ← i

end         t ← S[mp][2]; S[mp] ← S[i];
            itmp;  mp ← t

‘           i ← i+1;
            S[i] ← proc (S[mp][1]+1, S[mp][3], k)
            k ← P[k][2]; goto  T

value       if ¬ isr S[i] then goto A;
            S[i] ← S[i].;
            A: if ¬ isp S[i] then goto B;
            fct ← 0; t ← S[i];
            S[i] ← (bln t, mix t, mp, (), k);          (a Mark)
            mp ← i; k ← adr t; B:
```

72

```
call          i ← i-1;
              if ¬isr S[i] then goto A;
              S[i] ← S[i].;
              A: if lisp S[i] then goto ERROR;
              fct ← 0; t ← S[i];
              S[i] ← (bln t, mix t, mp, S[i+1], k);        (a Mark)
              mp ti; k ← adr t

  ,           k ← S[mp][5]; t ← S[mp][2];
              S[mp] ← S[i];
              itmp; mp ← t

goto          if ¬ isl S[i] then goto ERROR;
              mp ← mix S[i]; pp ← adr S[i];
              i ← mp; goto T

  )           t ← P[k][2]; s ← ()                           (build a list)
              A: if t = 0 then goto B;
              t ← t-1; s ← s & (S[i-t]); goto A;
              B: i ← i+1; i ← i - P[k][2];
              S[i] ← s
```

Certain features of ALGOL are not included in EULER, because they were
thought to be non basic (or not necessary), or because they did not fit
easily into the EULER definition, or both.
Examples are

the empty statement, allowing an extra semicolon before end,

the declaration list, avoiding the necessity of repeating the
declarator in front of each identifier,

the conditional statement without else,

the for-statement,

the own type.

It is felt that these features could be included in a somewhat
'fancier' EULER+ language, which is transformed into EULER by a prepass
to the EULER processing system. This prepass might include other features
- like 'macros' or 'clichés', it would take care of the proper deletion of
comments,etc. Certain standard macros or procedures might be known to
this prepass and could thus be used in EULER+ without having been declared,
like the standard functions in ALGOL. The set of these procedures would
necessarily have to include a complete set of practical input-output pro-
cedures. It should be noted, however, that in contrast to ALGOL, they can
be described in EULER itself, assuming the existence of appropriate opera-
tors in and out (reading and editing characters) of   s y m b o l s
and lists (formats are lists of symbols), of type-test- and conversion-
operators are of course instrumental in the design of these procedures.
A few other useful 'standard procedures' are given as programming exam-
ples in the following paragraph. (cf. 'for', 'equal' and 'array')

C. Examples of Programs

A list can contain elements of various types, here numbers and procedures:

```
begin new x; new s;
    s ← (2, `begin x ←x+1; s[x] end', `out x') x ← s[1]; s[x]
end
```

                * * * * * * * * *

A reference can be used to designate a **sublist**. Thus repeated double indexing is avoided:

```
begin new a; new r;
    a ← (1,(2,3),4);                    The output is: 2, 3
    r ←@a[2];
    out r.[1]; out r.[2];
    r.[1] ←Ω
end
```

                * * * * * * * * *

A procedure assigned to a variable (here p) is replaced by a constant, as soon as further execution of the test n < 100 is no longer needed:

```
begin new p; new n; new f;
    n ← 0;
    p ← `n ← n+1; if n < 100 then f(n) else p ← f(n)';
    f ← `formal x;  . . . . . . . . .......';
        . . . . . . . . . . . . . . . . . . . . . .
end
```

                * * * * * * * * *

If a parameter is a `value-parameter', the value is established at call time.  In the case of a `name-parameter', no evaluation takes place at call time. Thus the output of the following program is 4,16,3

```
begin new p; new a; new i;
     p ← 'formal x; formal k;
          begin k ← k+1; out x end';
     i ← 1;
     a ← (4, 9,16);
     p(a[i], @i); p('a[i]', @i); out i
end
begin new p; new a; new i;
     p ← 'formal x; formal k;
          begin k ← k+1; x ← k end';
     a ← list 3; i ← 1;
     p(@a[i], @i); p( '@a[i]', @i)
end
```

Here the final value of a is (2, Ω,3).

* * * * * S t * * * * *

A for statement is not provided in EULER. It can, however, easily be pro-
grammed as a procedure and adapted to the particular needs. Two examples
are given below, the latter corresponding to the ALGOL for:

```
for ← 'formal v; formal n; formal s;
          begin label k; v ← 1;
               k: if v ≤ n then
                    begin s; v ← v+1; goto k end
                    else Ω
     e n d'
algolfor ← 'formal v; formal l; formal step; formal u; formal s;
               begin label k; v ← l;
                    k: if (v-u) X step ≤ 0 then
                    begin s; v ← v + step; goto k end
                    else Ω
               end'
```

It should be noted that the decision whether the iterated statement should be able to alter the values of the increment and limit is made in each call for 'for' individually by either enclosing the actual parameters in quotes (name-parameter), or omitting the quotes (value-parameter).

E.g. a) n ← 5; for (@i, n, 'begin n ← n-1; out n end')

     b) n ← 5; for (@i, 'n', 'begin n tn-1; out n end')

a) yields 4,3,2,1,0, while b) yields 4,3,2 .

* * * * * * * * *

There is no provision for an operator comparing lists in EULER. But list comparisons can easily be programmed. The given example uses the 'for' defined above:

```
    equal ← 'formal x; formal y;
        begin new t; new i; label k;
            t ← false;
            if isli x ∧ isli y ∧ length x = length y then
                begin for (@i, length x,
                        'if ¬equal (@x[i], @y[i]) then goto k else Ω');
                    t ttrue
                end else
            t tisn x A isn y ∧ x=y;
        k: t
        end'
```

It should be noted that the definition of A deviates from **ALGOL** and thus makes this program possible; therefore in

     t tisn x ∧ isn y ∧ x=y

the relation **x=y** is never evaluated if either x   or y is a number. If the list elements may also be logical values or symbols, then the above statement must be expanded into:

t ← isn x A <u>isn</u> y A x=y ∨ <u>isb</u> x A <u>isb</u> y ∧ <u>real</u> x = real y ∨

    <u>isy</u> x A isy y A real x = real y

$$* \quad * \quad * \quad * \quad * \quad * \quad * \quad * \quad *$$

There is no direct provision for an array declaration (or rather array

'reservation') either. It can be programmed by the following procedure:

    array ← '<u>formal</u> ℓ; <u>formal</u> x;

        <u>begin</u> <u>new</u> t; <u>new</u> a; <u>new</u> b; <u>new</u> i;

            b ← ℓ; t ← <u>list</u> b[1];

            a ← <u>if</u> <u>length</u> b > 1 then array (tail b, x) else x;

            for (@i, b[1], 't[i] ← a');

            t

        <u>end</u>'

The statement a ← array ((xl, x2, . . . , xn)) would then correspond to

the ALGOL array declaration

                <u>array</u> a[1: xl, 1: x2, . . . , 1: xn],

while the statement a ← array ((xl, x2, . . . , xn), a) would additionally

'initialize all elements with α .

$$* \quad * \quad * \quad * \quad * \quad * \quad * \quad * \quad *$$

The following is an example of a summation procedure, using what is

in **ALGOL** known as 'Jensen's device'. The statement sum ('t', @i, I, u)

has the meaning of $\sum\limits_{i=\ell}^{u} t$.

    <u>begin new</u> k; <u>new</u> I; <u>new</u> sum; <u>new</u> a; <u>new</u> b;

        sum ← '<u>formal</u> t; <u>formal</u> i; <u>formal</u> ℓ; <u>formal</u> u;

            <u>begin</u> i ← ℓ;

                <u>if</u> ℓ > u ℓ+1, u)<u>else</u> t + sum ('t', @i,

            <u>end</u>';

    a ← (1, 4, 9,16);

    b ← ((1, 4), (9, 16));

```
        out sum ('a[k]', @k, 1, 4);
        out sum ('a[k] × a[5-k]', @k, 1, 4);
        out sum ('sum ('b[k][ℓ]', @ℓ, 1, 2)', @k, 1, 2)
    end


                * * * * * * * * * *


    begin new x; new sqrt; new elliptic; label K;
         elliptic ← 'formal a; formal b;
                    if abs [a-b] ≤ 10⁻6 then 1.570796326/a else
                    elliptic ([a+b]/2, sqrt (a×b))';
         sqrt ← 'formal a;
                begin label L; new x; x ← a/2;
                    L:if abs [x↑2 - a] < 10⁻8 then x else
                        begin x ← [x+a/x]/2; goto L
                        end
                end';
          x ← 0.7;
    K: out x; out sqrt(x); out elliptic (1,x);
        x ← x+0.1; if x ≤ 1.3 then goto K else Ω
    end
```

This program contains a square-root procedure using Newton's method
iteratively, and a procedure computing the elliptic integral

$$\int_0^{\frac{\pi}{2}} \frac{dt}{\sqrt{a^2\cos^2 t + b^2\sin^2 t}}$$

using the Gaussian method of the arithmetic-geometric mean recursively.

                * * * * * * * * *

As a final example, a permutation generator is programmed in EULER, so
that the value of

                        perm (1, ℓ)

79

is the list of all permutations of the elements of list l, i.e. a list with
1 x 2 X 3 x . . . x length l lists :

```
begin new perm; new a; new k; label f;
    perm ← 'formal k; formal y;
            begin  new  rot; new exch; new x;
                x ← y;
                rot ← 'formal k; formal m;
                        if m > length x then () else
                        perm (k+1, exch (k, m, @x)) & rot (k, m+1)';
                exch ← 'formal k; formal m; formal x;
                        begin new b; new t;
                            t tx;
                            b ← t[k]; t[k] ← t[m]; t[m] ← b; t
                        end';
                if length x = k then (x) else rot (k, k)
            end';
    a ← 0;
    f: out perm (1, a); a ← a & (length a); goto f
end
```

This program generates the following lists:
```
()
((0))
((0,1), (1,0))
((0,1,2), (0,2,1), (1,0,2), (1,2,0), (2,1,0), (2,0,1))
. . . . . . . . . . . . . . . . . . . . . . .
```

References

1. P. Naur, ed., 'Report on the algorithmic language ALGOL 60',
   Comm. ACM, vol. 3, pp. 299-314; (May 1960).

2. ----, 'Revised Report on the algorithmic language ALGOL 60',
   Comm. ACM, vol. 6, pp. 1-17 (Jan. 1963).

3. C. Böhm, 'The CUCH as a formal and descriptive language', IFIP
   Working Conf., Baden, Sept. 1964.

4. P. Landin, 'The mechanical evaluation of expressions', Comp. J.
   vol. 6, 4 (Jan 1964).

5. P. Landin, 'A correspondence between ALGOL 60 and Church's Lambda-
   Notation', Comm. ACM vol. 8, 2 and 3, pp. 89-101 and 158-165.

6. A. Church, 'The calculi of lambda-conversion', Ann. of Math. Studies,
   vol. 6, Princeton N. J., 1941.

7. A. Curry, R. Feys and W. Craig, 'Combinatory Logic', North-Holland
   Pub., 1958.

8. A. van Wijngaarden, 'Generalized ALGOL', Annual Review in Automatic
   Programming, vol. 3, pp. 17 -26.

9. A. van Wijngaarden, 'Recursive definition of syntax and semantics',
   IFIP Working Conf., Baden, Sept. 1964.

10. J. van Garwick, 'The definition of programming languages by their
    compiler', IFIP Working Conf., Baden, Sept. 1964.

11. R. W. Floyd, 'The syntax of programming languages - a survey',
    IEEE Trans. on El. Comp., vol. EC-13, pp. 346-353 (Aug. 1964).

12. Y. Bar-Hillel, M. Perles and E. Shamir, 'On formal properties of
    simple phrase structure grammars', Zeitschrift für Phonetik,
    Sprachwissenschaft und Kommunikationsforschung, vol. 14, pp. 143-172;
    also in 'Language and Information' Addison-Wesley Pub., 1964.

13. R. W. Floyd, 'A descriptive language for symbol manipulations', J. ACM, vol. 8, pp. 579-584 (Oct. 1961).

14. R. W. Floyd, 'Syntactic analysis and operator precedence', J. ACM, vol. 10, pp. 316-333 (July 1963).

15. E. T. Irons, 'Structural connections in formal languages', Comm. ACM, vol. 7, pp. 67-71 (Feb. 1964).

16. N. Wirth, 'A generalization of ALGOL', Comm. ACM vol. 6, pp. 547-554 (Sept. 1963).

Appendix I

The following is a listing of the syntax-processor programmed in Extended

ALGOL* for the Burroughs B5500 computer. The organization of this program

is summarized as follows:

Input lists of non-basic symbols, basic symbols and productions

A.

  B1.

    C1. Build list of leftmost and rightmost symbols, cf. III B2.

    C2. Establish precedence relations, cf. III B2.

  B2.   Find precedence functions, cf. III B5.

  B3.   Build tables to be used by the parsing algorithm of the
EULER processor. (punch cards)

    Most of the program is written in ALGOL proper. Often used extensions

of ALGOL are:

1. READ and WRITE statements
(symbol strings enclosed in < and > denote a format)

2. DEFINE declarations, being macros to be literally expanded by the
ALGOL compiler.

3. STREAM procedures, being B5500 machine-code procedures, allowing the
use of the B5500 character mode.

---

* cf. Burroughs B5500 Extended ALGOL Reference Manual.

```
BEGIN COMMENT SYNTAX-PROCESSOR.   NIKLAUS WIRTHDEC.1964J
     DEFINE NSY =150#J   COMMENT MAX.  NO. OF SYMBOLSJ
     DEFINE NPH = 150#J  COMMENT MAX. NO. OF  PRODUCTIONSJ
     DEFINE UPTO = STEP 1 UNTIL #J
     DEFINE LS = "<" #,  EQ = "=" #,  GR = ">" #,NULL="" #J
     FILE OUT PCH 0 (2,10)J    COMMENT  PUNCH  FILEJ
     INTEGER LTJ COMMENT NUMBER OF LAST NONBASIC SYMBOLJ
     INTEGER K,M,N,  MAX,  OLONJ BOOLEAN ERRORFLAGJ
     ALPHA  ARRAY  READBUFFER[0:9], WRITEBUFFER[0:14]J
     ALPHA  ARRAY TEXT [0:11]J  COMMENT  AUXILIARY TEXT ARRAYJ
     ALPHA  ARRAY  SYTB [0:NSY]J  COMMENT SYMBOLTABLEJ
     INTEGER  ARRAY  REF [0:NPR,0:5]J  COMMENT SYNTAX REFERENCE TABLEJ
     LABEL  START,EXITJ
     LABEL  A,B,C,E,F,GJ

STREAM PROCEOURE  CLEAR  (D,N)J  VALUE NJ
     BEGIN  01  + DJ  OS + 8  LIT  ""J SI t DJ DS t N WDS
     END J'
STREAM  PHOCEOURE  MARK  (D,S)J VALUESJ
     BEGIN   DI + DJSI t LOC SJ  SIt SI+7J DSt C H R
     END J
BOOLEAN   STREAM PROCEDUREFINIS(S)J
     BEGIN  TALLY  +1J SI t SJ IF  SC = ".." THEN  FINIS + TALLY
     END J
STREAM PROCEDURE  EDIT  (S,D,N)J
     BEGIN DI +   DJ  SI t NJ OS  t 3 DECJ SI t SJ DS + 9 WDSJ
     END J
STREAM PROCEDURE MOVE (S,D)J
     BEGIN  SI + SJ 01 t DJ OS + WDSJ
     END J
STREAM  PROCEDURE MOVETEXT(S,D,N)J VALUE NJ
     BEGIN   DI t DJ SI + SJ OS + N WDSJ
     END J
BOOLEAN STREAM PROCEDURE EQUAL (S,D)J
     BEGIN SI t SJ DI t DJ TALLY  + 1J IF 8SC = DC THEN EQUAL +TALLYJ
     END J
STREAM PROCEDURE  SCAN (S,DD,N)J
     BEGIN LABEL A,B,C,D,EJ
          SI t SJ DI t ODJ  DS t 48 LIT   "0"J DI + DDJ SI t SI+1J
          IF  SC = " " THEN DI t DI+8J
     AJ  IF  SC = " " THEN  BEGIN  SI +SI+1J GO TO A END J
          IF SC > "9" THEN GO TO DJ
          8 (IF SC="" THEN  BEGIN DS+LIT""J GO TO E END   J DS+CHRJEJ))J
     BJ  IF  SC # " " THEN  BEGIN  SI +SI+1J GO TO B END J
     CJ  SI t SI+1J  GO TO  AJ
     DJ  DI + NJ  SI +SI+5J OS + 3 OCT
     END J
STREAM PROCEDURE  EDITTEXT (S,D,N)J  VALUE NJ
     BEGIN   SI + SJ 01 t DJ DI+ DI+10J N(DI +DI+2J DS +  8  CHR)
     END J
STREAM PROCEDURE  SETTEXT (A,B,C,D,E,Z)J
     BEGIN   01  +ZJ 01 t DI+8J SI t AJ DS+ 3 DECJ SI +8J DS + WDSJ
          DI t DI+5J SI t CJ OS +3DECJ DI + DI+3J  SI  +DJ DS + 3 DECJ
          01 t DI+3J SI t EJ OS + 3 OECJ
```

84

```
            END J
STREAM  PROCEDURE  PCHTX(S,D,N)J  VALUE  N)
      BEGIN    SI ← S)  DI ← D)DI←DI+4)
          N(DS ← LIT  """) DS ← 8  CHR;  DS ← LIT  """) DS ← LIT  ",")J
      END J
PROCEDURE  INPUT)
      READ(CARDFIL,10, READBUFFER[*])  [EXIT])
PROCEDURE  OUTPUT)
      BEGIN WHITE (PRINFIL, 15, WRITEBUFFER[*]))
            CLEAR (WRITEBUFFER[0], 14))
      END J
INTEGER  PROCEDURE  IN% (X)) REAL  X)
      BEGIN INTEGER II LABEL F)
          FOR  1 ← 0 UPTO M  DO
              IF  EQUAL (SYTB[I], X)  THEN  GO  TO  F  J
          WRITE(<"UNDEFINED  SYMBOL">)) ERRORFLAG ←  TRUE)
      F) INX ←    Ii
      END)


START)
      FOR N ← 0 UPTO 5 DO
      FOR M ← 0 UPTO NPR  DO  REP  [M,N] ← 0)
      M ← N ← MAX ← OLDN ← 0)  ERRORFLAG  ← FALSE)
      CLEAR  (WRITEBUFFER[0],14))
      COMMENT READ LIST  OF  SYMBOLS,  ONE SYMBOL MUST  APPEAR PER CARD,
      STARTING IN COL,9 (8 CHARS,  ARE  SIGNIFICANT), THE LIST OF NON"
      BASIC SYMBOLS IS FOLLOWED BY  AN  ENDCARD ("*" IN COL,1), THEN
      FOLLOWS  THE LIST OF BASIC  SYMBOLS  AND AGAIN AN ENDCARD J
      WRITE (< "NONBASIC SYMBOLS)">))
A) INPUTJ
      IF  FINIS (READBUFFER[0]) THEN GO  TO  E)
      M ← M+1)
      MOVE (READBUFFER[1], SYTB [M,;
      EDIT (READBUFFER[0], WRITEBUFFER[1], M))
      OUTPUTJ  GO TO A)
E)  WRITE (</"BASIC SYMBOLS)">)) LT ← M)
F) INPUT)
      IF FINIS  (READBUFFER[0])  THEN  GO  TO G)
      M ← M+1)
      MOVE (READBUFFER[1], SYTB[M]))
      EDIT (READBUFFER[0], WRITEBUFFER[1], M))
      OUTPUTJ GO TO FJ

      COMMENT  READ THE LIST OF  PRODUCTIONS,  ONE PER CARD,  THE LEFTPART
      IS 4  NONBASIC SYMBOL STARTING IN COL,2, NO FORMAT IS  PRESCRIBED
      FOR THE RIGHT PART, ONE OR MORE BLANKS ACT 4  SSYMBOL SEPARATORS,
      IF COL,2 IS BLANK, THE SAME LEFTPART AS  IN  THE  PREVIOUS  PRODUCTION
      IS SUBSTITUTED. THE MAX.  LENGTH OF  A PRODUCTION IS 6 SYMBOLS)
G) WRITE (</"SYNTAX)">))
B) INPUT)
      IF FINIS (READBUFFER[0]) THEN GO TO C)
      MOVETEXT (READBUFFER[0], WRITEBUFFER[1], 10)) OUTPUT)
      MARK (READBUFFER[9], 12)) SCAN (READBUFFER[0],TEXT[0],N))
      IF N S 0 OR N > NPR OR REF[N,0] ≠ 0  THEN
          BEGIN WRITE (<"UNACCEPTABLE TAG">)) ERRORFLAG t TRUE) GO TO B
```

85

```
        END J
    IF N > MAX  THEN  MAX  t  NJ
    COMMENT  THE  SYNTAX  IS  STORED  IN  REF,  EACH  SYMBOL  REPRESENTED  BY
        ITS  INDEX  IN THE SYMBOL-TABLEJ
    FOR  K  + 0  UPTO  5  DO  REF  [N,K]  + INX (TEXT[K])J
    IF  REF  [N,0]  = 0  THEN  REF  [N,0]  + REF  [OLDN,0]  ELSE.
    IF  REF  [N,0]  > LT  THEN
        BEGIN WHITE (<"ILLEGAL PRODUCTION">)J ERRORFLAG + TRUE END J
    OLDN  + NJ  GO  TO  BJ
C: IF ERRORFLAG  THEN  GO  TO  EXIT)
    N + MAXJ
    COMMENT  M  IS  THE  LENGTH  OF  THE  SYMBOL-TABLE,  N OF THE REF-TABLEJ


BEGIN  COMMENT  BLOCK  AJ
    INTEGER  ARRAY  H[0:M, 0:M]J   COMMENT  PRECEDENCE  MATRIXJ
    INTEGER  ARRAY  F, G[0:M]J  COMMENT  PRECEDENCE  FUNCTIONSJ
BEGIN  COMMENT  BLOCK  B1J
    INTEGER  ARRAY  LINX, RINX [0:LT]J  COMMENT  LEFT / RIGHT  INDICESJ
    INTEGER  ARRAY  LEFTLIST,RIGHTLIST[0:1022]J
BEGIN  COMMENT  BLOCK  C1 ,   BUILD  LEFT-  AND  RIGHT-SYMBOL  LISTSJ
    INTEGER  I,JJ
    INTEGER  SP, RSPJ  COMMENT  STACK- AND  RECURSTACK-POINTERSJ
    INTEGER  LP, RPJ   COMMENT  LEFT/RIGHT  LIST POINTERSJ
    INTEGER  ARRAY INSTACK [0:M]J
    BOOLEAN  ARRAY  DONE, ACTIVE [0:LT]J
    INTEGER  ARRAY  RECURSTACK, STACKMARK [0:LT+1]J
    INTEGER  ARRAY  STACK, [0:1022]J COMMENT  HERE  THE  LISTS  ARE  BUILTJ

PROCEDURE  PRINTLIST (LX,L)J  ARRAY  LX, L [0]J
    BEGIN  INTEGER  I,J,KJ
        FOR  I   + 1 UPTO  LT  DO IF DONE[I]   THEN
        BEGIN  K  +0J  MOVE  (SYTB[I], WRITEBUFFER[0])J
            FOR  J + LX[I],J+1  WHILE  L[J] ≠ 0 DO
                BEGIN  MOVE  (SYTB[L[J]], TEXT[K])J  K + K+1J
                    IF  K ≥ 10 THEN
                    BEGIN  EDITTEXT (TEXT[0], WRITEBUFFER[0],10)J OUTPUTJ
                        K + 0 J
                    END J
                END J
            IF K > 0  THEN
            BEGIN  EDITTEXT(TEXT[0], WRITEBUFFER[0], K)J  OUTPUT  END J
        END
    END J
PROCEDURE  DUMPITJ
    BEGIN INTEGER I,JJ  WRITE ([PAGE])J
        WRITE (<X9,"DONE ACTIVE LINX RINX">)J
        WRITE (<5I6>, FOR I + 1 UPTO LT DO
                    [I,  DONE[I], ACTIVE[I], LINX [I], RINX[I]])J
        WRITE (</"STACK:     se =",I3>, SP)J
        WRITE (<I10,"  ",10I6>, FOR I + 0 STEP  10 UNTIL SP DO
            [I, FOR J  + I UPTO I+9 DO STACK [J]])J
        WRITE (</"RECURSTACK:">)J
        WRITE (<3I6>, FOR I + 1 UPTO  RSP  DO
                    [I, RECURSTACK[I], STACKMARK[I]])J
```

86

```
        END J
PROCEDURE  KESET (X)J VALUE XJ  INTEGER  X;
     BEGIN INTEGER  Ii
        FOR I + X UPTO  RSP  DO  STACKMARK  [I] + STACKMARK  [X]J
     END J
PROCEDURE   PUTINTOSTACK (X)J VALUE XI  INTEGER  XJ
     COMMENT  X Is PUT INTO THE WORKSTACK. DUPLICATION IS AVOIDED!
     BEGIN IF  INSTACK [X] = 0 THEN
           BEGIN SP + SP+1J STACK  [SP] + XJ INSTACK [XI + SP  END
           ELSE IF INSTACK [X] < STACKMARK  [RSP] THEN
           BEGIN SP + SP+1J STACK  [SP] + XJ
              STACK  [INSTACK[X]] + OJ INSTACK [X] + SPJ
           END J
        IF SP > 1020  THEN
        BEGIN  WRITE  (</"STACK OVERFLOW"/>)J DUMPITJ GO  TO  EXIT  END J
     END J
PROCEDURE   COPYLEFTSYMBOLS (X)J VALUE XJ INTEGER Xi
     'COMMENT  COPY  THE  LIST  OF  LEFTSYMBOLS  OF X INTO  THE  STACKJ
     BEGIN  FUR  X + LINX[X], X+1  WHILE  LEFTLIST[X] # 0  DO
           PUTINTOSTACK  (LEFTLIST[X])J
     END J
PROCEDURE   COPYRIGHTSYMBOLS (X)J VALUE XJ  INTEGER  XJ
     COMMENT  COPY  THE  LIST  OF  RIGHTSYMBOLS  OF X INTO THE STACKJ
     BEGIN  FUR  X +RINX[X], X+1  WHILE  RIGHTLIST[X] # 0  DO
           PUTINTOSTACK  (RIGHTLIST[X])J
     END J
PROCEDURE   SAVELEFTSYMBOLS  (X)J VALUE Xi  INTEGER  XI
     COMMENT  THE  LEFTSYMBOLLISTS  OF  ALL  SYMBOLS  IN  THE  RECURSTACK
     WITH INDEX > X HAVE 'BEEN BUILT  AND  MUST  NOW BE  REMOVED,  THEY  ARE
     COPIED INTJ "LEFTLIST" AND  THE  SYMBOLS  ARE  MARKED  "DONE"  J
     BEGIN INTEGER  I,J,UJ LABEL  L,EXJ
     LI IF  STACKMARK CXJ = STACKMARK  [X+1]  THEN
        BEGIN X + X+1J IF X < RSP  THEN  GO  TO  L  ELSE  GO  TO  LX END  J
        STACKMAHK  [RSP+1]+ SP+1J
        FOR I  + X+1 UPTQ RSP DO
        BEGIN  LINX [RECURSTACK[I]]+ LP+1J
           ACTIVE  [RECURSTACK[I]] + FALSEJ  DONE  [RECURSTACK[I]]+ TRUEJ
           FUR  J + STACKMARK[I] UPTO STACKMARK[I+1]-1  DO
           IF  STACK [J] # 0  THEN
           BEGIN LP + LP+1JLEFTLIST [LP] +  STACK  [J]J
           If LP > 1020  THEN
           BEGIN WRITE  (</"LEFTLIST OVERFLOW"/>)J DUMPITJ
                   PRINTLIST  (LINX, LEFTLIST)J GO  TO  EXIT
           END J
           END
        END J
        LP + LP+1J LEFTLIST [LP] + OJ
     EXIRSP + XJ
     END J
PROCEDURE   SAVERIGHTSYMBOLS   (X)J VALUE XJ  INTEGER  XJ
     COMMENT  ANALOG  TO  "SAVELEFTSYMBOLS"J
     BEGIN  INTEGER  I,JJ LABEL  L,EXJ
     LI I  F STACKMARK [X] = STACKMARK  [X+1]  THEN
        BEGIN  X + X+1J IF X < RSP  THEN  GO  TO  L  ELSE  GO  TO  EX  END  J
        STACKMAHK  [RSP+1] + SP+1J
```

```
        FOR I ← X+1   UPTO   RSP   DO
        BEGIN RINX [RECURSTACK[I]] ← RP+1J
            ACTIVE [RECURSTACK[I]]← FALSE; DONE [RECURSTACK[I]] t  TRUEJ
            FUR J ← STACKMARK[I] UPTO STACKMARK[I+1]-1  DO
            IF STACK [J] ≠ 0  THEN
            BEGIN RP←RP+1J RIGHTLIST [RP] • STACK  [J]J
            IF RP > 1020  THEN
            BEGIN WRITE  (</"RIGHTLIST OVERFLOW"/>)J DUMPITJ
                  PRINTLIST (RINX,RIGHTLIST)J GO TO EXIT
            END J
            END
        END J
        RP  • RP+1J RIGHTLIST [RP]← 0J
    EXIRSP ← XJ
    END J
PROCEDURE  BUILDLEFTLIST  (X)J VALUE Xi  INTEGER XJ
    COMMENT THE LEFTLIST OF THE SYMBOL X IS BUILT BY SCANNING THE
    SYNTAX FORPRODUCTIONS WITH LEFTPART = X, THE LEFTMOST SYMBOL IN
    THE RIGHTPART IS THEN INSPECTED:  IF It IS NONBASIC AND NOT MARKED
    DONE, ITS LEFTLIST IS BUILT FIRST, WHILE A SYMBOL  IS BEING INSPECTED
    IT IS MARKEO ACTIVEJ
    BEGIN INTEGER I,R,OWNRSPJ
        ACTIVE[X] ← TRUEJ
        RSP ← OWNRSP ← LINX [X]← RSP+1J
        RECUHSTACK CRSPJ ← Xi  STACKMARK [RSP]← SP+1J
        FOR I ← 1 UPTO N DO
        IF REF [I,0] = X THEN
        BEGIN IF OWNRSP < RSP THEN SAVELEFTSYMBOLS (OWNRSP)J
            R ← REF[I,1]J PUTINTOSTACK  (R)J
            If R ≤ LT   THEN
            BEGIN IF DONE [R] THEN COPYLEFTSYMBOLS CR)  ELSE
                IF ACTIVE[R] THEN RESET (LINX [R]) ELSE
                BUILDLEFTLIST (R)J
            END
        END J
    END J
PROCEDURE  BUILDRIGHTLIST(X)J VALUE XJ INTEGER XJ
    COMMENT  ANALOG TO "BUILDLEFTLIST"J
    BEGIN INTEGER I,R,OWNRSPJ LABEL QQJ
        ACTIVE [X] ← TRUE;
        RSP ← OWNRSP ← RINX [X]← RSP+1J
        RECUKSTACK [RSP]← Xi  SJACKMARK [RSP]← SP+1J
        FOR I ← 1 UPTO  N  DO
        If  REF [I,0]= X THEN
        BEGIN  IF OWNRSP < RSP  THEN SAVERIGHTSYMBOLS (OWNRSP)J
            FOR R ← 2,3,4,5  DO If  REF [I,R]≠0  THEN GO TO QQJ
        QQI R ← REF [I,R-1]J  PUTINTOSTACK  (R))
            If R ≤LT   THEN
            BEGINIF DONE  [R]  THEN COPYRIGHTSYMBOLS(R) ELSE
                IF ACTIVE [R] THEN RESET (RINX[R])ELSE
                BUILDRIGHTLIST (R)J
            END
        END
    END J
```

88

```
    SP ← RSP ← LP ← 0;
    FOR I← 1  UPTU  LT  DO  DONE[I] ←  FALSE;
    FOR  I←1 UPTO LT DO  IF  NOT  DONE  [I]  THEN
    BEGIN  SP ← RSP ←0;
        FOR     J    ←1 UPTO M DO INSTACK [J] ← 0;
        BUILDLEFTLIST  (I); SAVELEFTSYMBOLS  (0);
    END ;
    WRITE  ([PAGE]); WRITE  (<X20,"***  LEFTMOST  SYMBOLS  ***"/>);
    PRINTLIST  (LINX, LEFTLIST);
    SP ← RSP ← HP ← 0;
    FOR  I← 1 UPTO LT  00  DONE[I] ← FALSE;
    FOR  I  ← 1  UPTO LT DO  IF  NOT  OONE  [I]  THEN
    BEGIN  SP ← RSP ← 0;
        FOR  J ← 1  UPTO M 00  INSTACK [J] ← 0;
        BUILDRIGHTLIST (I);  SAVERIGHTSYMBOLS (0);
    END ;
    WRITE  ([3]);       WRITE  (<X20,"***  RIGHTMOST  SYMBOLS  ***"/>);
    PRINTLIST (RINX, RIGHTLIST);
END  BLOCK  C1;


BEGIN  COMMENT  BLOCK  C2,   BUILD  PRECEDENCE  RELATIONS;
        INTEGER  J,K,P,Q,R,L,T;
        LABEL  NEXTPRODUCTION;
PROCEDURE   ENTER  (X,Y,S);  VALUE  X,Y,S;  INTEGER  X,Y,S;
    COMMENT  ENTER  THE  RELATION  S INTO POSITION  [X,Y],  CHECK  FOR  DOUBLE-
    OCCUPATION  OF  THIS  POSITION;
    BEGIN      T ←H[X,Y]; If T≠  NULL   AND T≠S  THEN
        BEGIN  ERRORFLAG  t  TRUE;
        WRITE  (<"PRECEDENCE  VIOLATED  BY  ",2A1," FOR PAIR",2I4,
                "  BY  PRODUCTION",I4>,  T,  S,  X,  Y,  J);
        END  J
        H[X,Y]←  St
    END  J
    WRITE  ([PAGE]);
    FOR  K ←1 UPTO  M  DO
    FOR  J ←  1  UPTD  M  DO H[K,J] ←  NULL;
    FOR  J← 1  UPTO  N  DO
    BEGIN   FUR  K ← 2,3,4,5  00  IF REF [J,K] ≠ 0  THEN
        BEGIN  P ← REP [J,K-1]; Q ← REF [J,K];
            ENTER  (P,Q,EQ);
            If P  S  LT   THEN
            BEGIN FOR R←RINX[P], R+1 WHILE RIGHTLIST [R]≠ 0 DO
                    ENTER  (RIGHTLIST[R],Q,GR);
                If Q S LT   THEN
                FOR L←LINX[Q],L+1 WHILE LEFTLIST [L] ≠ 0  00
                BEGIN ENTER (P, LEFTLIST [L], LS);
                    FOR R←RINX[P],R+1 WHILE RIGHTLIST [R] ≠ 0  DO
                        ENTER (RIGHTLIST[R],LEFTLIST[L],GR)
                END
            END
            ELSE   IF Q S LT  THEN
                FUR L←LINX[Q],L+1 WHILE LEFTLIST [L] ≠ 0  00
                    ENTER  (P, LEFTLIST[L],LS);
        END
```

89

```
          ELSE  GO  TO  NEXTPRODUCTION
     NEXTPRODUCTION  END J  ;
     WRITE  (</X3,39I3/>, FOR J  ←1 UPTO M  00  J);
     FOR  K  ←1 UPTO M  DO
     WRITE  (<I3,39(X2,A1)>, K, FOR J ←  1  UPTO M  DO H[K,J])
END  BLOCK  C2
END  BLOCK  B1
     IF  ERRORFLAG  THEN  GO TO EXIT;
     WRITE  (</"SYNTAX  IS A PRECEDENCE   GRAMMAR"/>);


BEGIN  COMMENT  BLOCK  B2,   BUILD F AND G  PRECEDENCE  FUNCTIONS1
     INTEGER I,  J,  K, K1,  N,  FMIN,  GMIN,  T
PROCEDURE THRU (I,J,X)  VALUE I,J,X  INTEGER I,J,X
     BEGIN  WHITE  (</"NO PRIORITY  FUNCTIONS  EXIST  ",3I6>, I,J,X)
          GO TO  EXIT
     END
PROCEDURE  FIXUPCOL (L,J,X)  VALUE L,J,X  INTEGER L,J,X   FORWARD;
PROCEDURE  FIXUPROW(I,L,X)  VALUE I,L,X  INTEGER I,L,X
     BEGIN  INTEGER J F[I]←G[L]←X
          IF  K1 ≠ K  THEN
          BEGIN IF H[I,K]= EQ  AND  F[I] ≠G[K]  THEN  THRU  (I,K,0)   ELSE
               IF H[I,K]= LS  AND F[I] ≥G[K]  THEN   THRU  (I,K,0)
          END
     FOR  J  ←K1 STEP -1  UNTIL  1 00
          IF H[I,J]= EQ  AND  F[I] ≠G[J]  THEN  FIXUPCOL (I,J,0) ELSE
          If H[I,J] = LS  AND  F[I] ≥ G[J]  THEN  FIXUPCOL(I,J,1)
     END
PROCEDURE f  IXUPCOL (L, J,X)  VALUE L, J,X  INTEGER L, J,X
     BEGIN  INTEGER I G[J] ← F[L] + X
          IF  K1≠ K  THEN
          BEGIN IF H[K,J] = EQ  AND  F[K] ≠G[J]  THEN  THRU(K,J,1) ELSE
               IF H[K,J] = GR  AND  F[K] ≤ GCJJ THEN   THRU  (K,J,1)
          END
     FOR  I ← K  STEP  -1  UNTIL  1  DO
          IF H[I,J] = EQ  AND  F[I] ≠G[J]  THEN  FIXUPROW(I,J,0)ELSE
          IF H[I,J] = GR  AND  F[I] ≤G[J]  THEN  FIXUPROW(I,J,1)
     END
     K1 t  0
     FOR  K  ←1 UPTO M  DO
     BEGIN  FMIN ←1
          FOR  J ← 1 UPTO K1  DO
               IF H[K,J] = EQ  AND  FMIN < G[J]  THEN  FMIN ← G[J]  ELSE ,
               IF H[K,J]= GR  AND  FMIN≤ G[J]  THEN  FMIN t  G[J]+1
          F[K] ← FMN;
          FOR  J ←K1  STEP  -1 UNTIL  1 DO
               IF H[K,J] = EQ  AND  FMIN > G[J]  THEN  FIXUPCOL  (K,J,0)  ELSE
               IF H[K,J] = LS  AND  FMIN ≥G[J]  THEN  FIXUPCOL  (K,J,1)
          K1 ← K1+1  GMIN ← 1
          FOR  I ← 1 UPTO  K  00
               IF H[I,K]= EQ  AND  F[I]> GMIN  THEN  GMN ←F[I]  ELSE
               IF H[I,K]= LS  AND F[I]≥ G M I N  THEN   GMN ←F[I]+1
          G[K] ← GMINJ
          FOR I ←  K  STEP   -1 UNTIL 1 D O
               IF H[I,K] = EQ  AND  F[I] <GMIN  THEN  FIXUPROW(I,K,0) ELSE:
```

90

```
                    IF H[I,K] = GR  AND  F[I] ≤ GMIN THEN FIXUPROW(I,K,1);
        END K ;
END BLOCK B2 ;
    WRITE ((PAGE));

BEGIN COMMENT BLOCK B3, BUILD TABLES Of PRODUCTION REFERENCES;
    INTEGER I,J,K,L;
    INTEGER ARRAY MTB [0:M];   COMMENT MASTER TABLE ;
    INTEGER ARRAY PRTB [0:t02211    COMMENT PRODUCTION TABLE ;
    L ← 0;
    FOR I ← 1 UPTO M DO
        BEGIN MTB[I]←L+1;
            FUR J ← 1 UPTO N DO
            IF REF[J,1] = I THEN
                BEGIN FOR K ← 2,3,4,5 DO
                    IF REF[J,K] ≠ 0 THEN
                        BEGIN 1. ← L+1; PRTB[L] ← REF[J,K]
                        END ;
                    L ← L+1; PRTB[L] ← -J; L ← L+1; PRTB[L] ← REF [J,0];
                END ;
            L   ← L+1; PRTB[L] ← 0
        END ;

    COMMENT PRINT AND PUNCH THE RESUTS;
    SYMBOLTABLE, PRECEDENCE FUNCTIONS, SYNTAX REFERENCE TABLES;
    WRITE (<X8,"NO.",X5,"SYMBOL",X8, "F",X5,"G",X4,"MTB"/>);
    FOR I ← 1 UPTO M DO
    BEGIN SETTEXT(I,SYTB[I],F[I],G[I], MTB[I], WRITEBUFFER[0]);
        OUTPUT
    END ;
    WRITE (</"PRODUCTION TABLE:"/>);
    FOR I ← 0 STEP 10 UNTIL L DO
    WRITE (<I9,X2,10I6>, FOR I ← 0 STEP 10 UNTIL I. DO
            [I, FOR J ← I UPTO I+9 DO PRTB[J]]);
    WRITE ((/"SYNTAX  VERSION ",A5>, TIME (0));
    WRITE (PCH, <X4,"FT←",I3,"; LT ←",I4,"; LP ←",I4,";">,LT+1,M,L);
    FOR I ← 1 STEP 6 UNTIL M DO
    BEGIN PCHTX (SYTB[I], WRITEBUFFER[0], IF M-I≥6 THEN 6 ELSE M-I+1);
        WRITE (PCH,10,WRITEBUFFER[*]); CLEAR (WRITEBUFFER[0],9)
    END ;
    WRITE (PCH, <X4,12(I4,",")>, FOR I ← 1 UPTO M 00 F[I]);
    WRITE (PCH, <X4,12(I4,",")>, FOR I ← 1 UPTO M DO G[I]);
    WRITE (PCH,<X4,12(I4,",")>, FOR I ← 1 UPTO M DO MTB[I]);
    WRITE (PCH,<X4,12(I4,",")>, FOR I ←1 UPTO L DO PRTB[I]);
END BLOCK B3
END BLOCK A ;

EXIT:
END,
```

Appendix II

The following is a listing of the EULER processing system programmed in
Extended ALGOL for the Burroughs B5500 computer. The organization of
this program is summarized as follows :

EULER Translator

Declarations including the procedure INSYMBOL and the code-generating

procedures P1, P2, P3, FIXUP,

Initialization of tables with data produced by the syntax-processor,

The parsing algorithm,

The interpretation rules (their labels correspond to their numbering

in IV B)

EULER Interpreter

Declarations including the procedures DUMPOUT (used for outputting

results) and FREE (used to recover no longer used storage space when

memory space becomes scarce)

The interpretation rules for the individual instructions

The source program is punched on cards (col. 1-72) in free field
format. Blank spaces are ignored, but may not occur within identifiers or
word-delimiters.

An identifier is any sequence of letters and digits (starting with
a letter), which is not a word-delimiter. Only the first 8 characters
are significant; the remaining characters are ignored.

A **word-delimiter** is a sequence of letters corresponding to a single
EULER symbol, which in the reference-language is expressed by the same
sequence of underlined or boldface letters.   E.g., begin → BEGIN,
end → END etc.   Note: ' → LQ,   ' → RQ,   ₁₀ → TEN, Ω → UNDEFINED.

A symbol is any **BCL-character*** (or sequence of up to 5 XL-characters)
enclosed between characters '"'. E.g. "*"

An example of an EULER program is listed at the end of this
Appendix.

---

* cf. Burroughs B5500 Extended ALGOL Reference Manual.

```
BEGIN COMMENT    E U L E R   I V   S Y S T E M           MARCH 1965 J
        INTEGER FT, LT) COMMENT INDEX OF FIRST AND LAST BASIC SYMBOL)
        INTEGER LP)   COMMENT LENGTH OF PRODUCTION TABLE)
        ARRAY PKOGKAM CO1102211
        DEFINE AFIELD = [39:9] #, BFIELD = [9:30] #, C F I E L D = [1:8] #)
        LABEL EXIT)
        FT + 45;  LT + 119) LP + 465)      COMMENT DATA GENERATED BY SY-PR.)

BEGIN COMMENT E U L E R   I V   TRANSLATOR          N.WIRTH )
        DEFINE MARK = 119 #, IDSYM = 63 #, REFSYM = 59 #, LABSYM = 62 #)
        DEFINE VALSYM = 56 #, CALLSYM = 55 #, UNDEF = 0 #, NEWSYM = 60 #)
        DEFINE UNARYMINUS = 116 #, NUMSYM = 68 #, BOOLSYM = 64 #)
        DEFINE LISTSYM = 1028, SYMSYM = 113 #, FORSYM = 61 #)
        DEFINE NAME = VCOJ #)
        INTEGER I,J,K,M,N,R,T,T1,SCALE) BOOLEAN ERRORFLAG)
        INTEGER BN, ON)   COMMENT BLOCK- AND ORDER-NUMBER)
        INTEGER NP) COMMENT NAME LIST POINTER )
        INTEGER MPJ     COMMENT MARK-POINTER OF NAME-LIST)
        INTEGER PRP)    COMMENT PROGRAM POINTER;
        INTEGER WC, CC) COMMENT INPUT POINTERS;
        ALPHA ARRAY READBUFFER, WRITEBUFFER[0:14])
        ALPHA ARRAY SYTB [0:LT]) COMMENT TABLE OF BASIC SYMBOLS)
        INTEGER ARRAY F, G [0:LT]) COMMENT PRIORITY FUNCTIONS )
        INTEGER ARRAY MTB [0:LT]) COMMENT SYNTAX MASTER TABLE )
        INTEGER ARRAY PRTB [0:LP]) COMMENT PRODUCTION TABLE)
        INTEGER ARRAY S [0:127]) COMMENT STACK )
        REAL ARRAY V [0:127]) COMMENT VALUE STACK )
        ALPHA ARRAY NL1 [0:63]) COMMENT NAME LIST )
        INTEGER ARRAY NL2, NL3, NL4 [0:63])
        LABEL A0,A1,A2,A3,A4,A5,A6,A7,A8,A9)
        LABEL LO, L1131, NAMEFOUND,
        L1,L2,L3,L4,L5,L6,L7,L8,L9,L10,L11,L12,L13,L14,L15,L16,L17,L18,L19,
        L20,L21,L22,L23,L24,L25,L26,L27,L28,L29,L30,L31,L32,L33,L34,
        L35,L36,L37,L38,L39,L40,L41,L42,L43,L44,L45,L46,L47,L48,L49,L50,L51,
        L52,L53,L54,L55,L56,L57,L58,L59,L60,L61,L62,L63,L64,L65,L66,L67,L68,
        L69,L70,L71,L72,L73,L74,L75,L76,L77,L78,L79,L80,L81,L82,L83,L84,L85,
        L86,L87,L88,L89,L90,L91,L92,L93,L94,L95,L96,L97,L98,L99,L100,L101,
        L102,L103,L104,L105,L106,L107,L108,L109,L110,L111,L112,L113,L114,
        L115,L116,L117,L118,L119,L120)
     SWITCH BRANCH +
        L1,L2,L3,L4,L5,L6,L7,L8,L9,L10,L11,L12,L13,L14,L15,L16,L17,L18,L19,
        L20,L21,L22,L23,L24,L25,L26,L27,L28,L29,L30,L31,L32,L33,L34,
        L35,L36,L37,L38,L39,L40,L41,L42,L43,L44,L45,L46,L47,L48,L49,L50,L51,
        L52,L53,L54,L55,L56,L57,L58,L59,L60,L61,L62,L63,L64,L65,L66,L67,L68,
        L69,L70,L71,L72,L73,L74,L75,L76,L77,L78,L79,L80,L81,L82,L83,L84,L85,
        L86,L87,L88,L89,L90,L91,L92,L93,L94,L95,L96,L97,L98,L99,L100,L101,
        L102,L103,L104,L105,L106,L107,L108,L109,L110,L111,L112,L113,L114,
        L115,L116,L117,L118,L119,L120)

     STREAM PROCEQURE ZERO (D))
        BEGIN DI+ D) OS + 8 LIT "0")
        END )
     STREAM PROCEDURE CLEAR (D))
        BEGIN    DI+D)OS+ 8 LIT " ")  SI + D) OS . 14 WDS
        END )
```

```
STREAM  PROCEDURE   MOVE (S,D)J
    BEGIN   SI  ← SJ DI←DJDS←   WDS
    END J
BOOLEAN  STREAM  PROCEDURE  EQUAL  (X,Y)J
    BEGIN   TALLY  ← 1;  SI ← XJ DI ←YJ IF 8SC ≡ DC THEN EQUAL ← TALLY
    END J


INTEGER  PROCEDURE  INSYMBOL;
    COMMENT  "INSYMBOL" READS  THE  NEXT  EULER-SYMBOL  FROM INPUT. 4
    STRINGS  OF  LETTERS  AND DIGITS ARE  RECOGNIZED AS  IDENTIFIERS, IF
    THEY  ARE  NUT  EQUAL  TO  AN  EULER-IVWORD-DELIMITER,
    A  CHARACTER-SEQUENCE  ENCLOSED  IN  " IS  RECOGNIZED  AS  A  SYMBOL;
    BEGIN  INTEGER  IJ LABEL  A,B,C,D,EJ
STREAM  PROCEDURE  TRCH  (S,M,D,N)J VALUE  M,NJ
    BEGIN   SI  ← SJ SI←SI+MJ DI ← DJ DI ← DI+NJ OS ← CHR
    END J
BOOLEAN  STREAM  PROCEOURE  BLANK  (S,N)J VALUE NJ
    BEGIN   TALLY  ←1JSI ← SJ SI ←SI+NJ IF SC ≡ " " THEN  BLANK  ←  TALLY
    END J
STREAM  PROCEDURE  BLANKOUT  (0);
    BEGIN    DI←DJ DS ← 8 LIT " "J
    END J
BOOLEAN  STREAM  PROCEDURE  QUOTE  (S,N)J VALUE  NJ
    BEGIN   TALLY  ← 1;  SI ←SJ SI ← SI+NJ IF SC ≡ """ THEN  QUOTE  ←  TALLY
    END J
BOOLEAN  STREAM  PROCEDURE  LETTER  (S,N)J VALUE  NJ
    BEGIN   TALLY  ← 1J SI ←  SJ SI ←SI+NJ
        IF SC ≡  ALPHA  THEN
        BEGIN IF SC < "0" THEN LETTER ← TALLY END
    END J
BOOLEAN  STREAM  PROCEDURE  LETTERORDIGIT  (S,N)J VALUE  NJ
    BEGIN    TALLY  ← 1;  SI ← SJ SI←SI+NJ
        IF SC ≡  ALPHA  THEN  LETTERORDIGIT ← TALLY
    END J
STREAM  PROCEOURE  EDIT (N, S, 0);  VALUE  NJ.
    BEGIN  SI ← LOC NJ DI ←DJ OS ← 3 OECJ
        SI ← SJ 01 ← 01 ←13J DS ← 10 WDS
    END J
PROCEDURE  ADVANCE;
    COMMENT  ADVANCES  THE  INPUT  POINTER BY 1 CHARACTER POSITIONJ
    BEGIN  IF CC ≡ 7  THEN
        BEGIN IF WC ≡ 8 THEN
            BEGIN  READ  (CARDFIL,10,READBUFFER[*])[EXIT]J
                EOIT (PRP+1, READBUFFER[0], WRITEBUFFER[0])J
                WRITE  (PRINFIL,15, WRITEBUFFER[*])J WC ← 0
            END ELSE WC ← WC+1J
            cc ← 0J
        END
        ELSE CC ← CC+1J
    END ADVANCE J
        BLANKOUT ( NAME);
A,  IF BLANK  (READBUFFER [WC], CC)  THEN
        BEGIN ADVANCEJ GO TO A END J
    IF  LETTER (READBUFFER [WC], CC)  THEN
        BEGIN FOR I ← 0 STEP 1 UNTIL 7 00
```

95

```
              BEGIN TRCH  (READBUFFER [WC], CC,  NAME,  I);   AOVANCEJ
                   If NOT  LETTERORDIGIT  (READBUFFER [WC], CC)  THEN  GO  TO  C
              END J
          B:  AOVANCEJ
              IF  LETTERORDIGIT  (READBUFFER [WC], CC)  THEN GO TO BJ
          C:
          END ELSE
      IF  QUOTE  (READBUFFER [WC], CC)  THEN
          BEGIN AOVANCEJ  ZERO  (NAME)J  NAME ← "       "J
          E: TRCH (READBUFFER[WC], CC,  I, 7); ADVANCE;
              IF I ≠ """    THEN
              BEGIN NAME ← I,[42:6] & NAME [18:24:24]J GO TO E END
              ELSE I← SYMSYMJ  GO TO 0
          END ELSE
          BEGIN TRCH  (REAOBUFFER [WC], CC,  NAME, 0)J  ADVANCE
          END J
      FOR I← FT STEP 1  UNTIL LT DO
       ·II; EQUAL (SYTB[I], NAME)  THEN  BEGIN ZERO(NAME)J GO TO 0 END J
      I ← IDSYMJ
  D:  INSYMBUL ← I
  END INSYMBOL J


PROCEDURE  P1(X)J  VALUE XJ  INTEGER XJ
  BEGIN  PRP ← PRP+1J PROGRAM[PRP] ← X
  END J
PROCEDURE  P2(X,Y)J  VALUE X,YJ  INTEGER X;REAL YJ
  BEGIN PRP ←PRP+1J PROGRAM[PRP] ← XJ PROGRAM[PRP],BFIELD ← YJ
  END J
PROCEDURE P3(X,Y,Z)J  VALUE X,Y,ZJ  INTEGER X,Y,ZJ
  BEGIN  PHP ←PRP+1J PROGRAM[PRP] ← XJ  PROGRAM[PRP],BFIELD ←YJ
     PROGRAM[PRP],CFIELD ← Z
  END J
PROCEDURE  FIXUP(I,X)J  VALUE I,XJ  INTEGER I,XJ
  PROGRAM[I],BFIELD ← XJ
PROCEOURE  ERROR  (N)JVALUE NJ  INTEGER NJ
  BEGIN  SWITCH FORMAT ERR ←
      ("UNDECLARED IDENTIFIER"),
      ("NUMBER TOO LARGE"),
      ("LABEL IS DEFINEDTWICE"),
      ("A LABEL IS NOT DECLARED"),
      ("LABEL DECLARED BUT NOT DEFINED?),
      ("PROGRAM SYNTACTICALLY INCORRECT");
      ERRORFLAG ← TRUE;
      WRITE  ([NO], ERR[N])J  WRITE (<X40,"COL.",I3>, WCX8 ← CC +1)
  END ERROR J


PROCEDURE  PROGRAMDUMPJ
  BEGIN REAL TJ  INTEGER IJ LABEL LJ
STREAM PROCEDURE  NUM (N,D)J  VALUE NJ
  BEGIN    01 ← DJ SI ←LOC NJ OS ← 3 OE C
  END J
          READ (<A4>, T) [L]J If T ≠  "DUMP"  THEN GO TO LJ
          WRITE(<//"PROGRAM DUMP">)J
          FOR I← 1 STEP 1 UNTIL  PRP  00
          BEGIN CLEAR (WRITEBUFFER[0])J
```

```
           T + PROGRAM[I]J  NUM (I, WRITEBUFFER[0])J
           MOVE (SYTB [T.AFIELD], WRITEBUFFER[1])J
           IF T.BFIELD ≠ 0 THEN NUM (T.BFIELD, WRITEBUFFER[2])J
           IF T.CFIELD ≠ 0 THEN NUM (T.CFIELD, WRITEBUFFER[3])J
           IF T.AFIELD = NUMSYM THEN
           BEGIN I +I+1J WRITE ([NO],<X14,E16.8>, PROGRAM[I]) END J
     WRITE (PRINFIL, 150 WRITEBUFFER[*])
     END J
L:END PROGRAMDUMP J
```

COMMENT INITIALISE THE **SYMBOLTABLE,** THE **PRIORITY** FUNCTIONS AND THE PRODUCTION TABLES WITH DATA GENERATED BY THE SYNTAX-PROCESSORJ

```
FILL  SYTB[*]WITH O n
"PROGRAM ",BLOCK     ",BLOKHEAD","BLOKBODY","LABDEF    ","STAT    ",
"STAT-    ",EXPR      ",EXPR-     ",IFCLAUSE","TRUEPART","CATENA   ",
"DISJ     ",DISJHEAD",CONJ     ",CONJ-     ",CONJHEAD","NEGATION",
"RELATION","CHOICE   ",CHOICE- ",SUM      ",SUM-     ",TERM     ",
"TERM-  ,  ",FACTOR   ",FACTOR- "#"PRIMARY  ",PROCDEF",PROCHEAD",
"LIST*    ",LISTHEAD","REFERENC","NUMBER   ",REAL*    ",INTEGER*",
"INTEGER-"#"DIGIT   ",LOGVAL   ",VAR      ",VAR-     ",VARDECL ",
"FORDECL  ",LABDECL",O         ",1       ",2       ",3       ",
"4        ",5        ",6        ",7       ",8       ",9       ",
",         ",.        ",J        ",I       ",@       ",NEW     ",
"FORMAL   ",LABEL    ",IDENT*   ",[       ",]       ",BEGIN   ",
"END      ",(        ",)        ",LQ      ",RQ      ",GOTO    ",
"OUT      ",+        ",IF       ",THEN    ",ELSE    ",&       ",
"OR       ",AND      ",NOT      ",=       ",≠       ",<       ",
"≤        ",≥        ",>        ",MIN     ",MAX     ",+       ",
"-         ",X        ",/        ",%       ",MOD     ",*       ",
"ABS      ",LENGTH   "#"INTEGER ",REAL    "#"LOGICAL ",LIST    ",
"TAIL     ",IN       ",ISB      ",ISN     ",ISR     ",ISL     ",
"ISLI     ",ISY      ",ISP      ",ISU     ",SYMBOL* ",UNDEFINE",
"TEN      ",#        ",TRUE     ",FALSE   ",$       "  J
FILL F[*] WITH 00
   1,    40    190    1,     20    1,    28    30    4,    1,    40    4,
  58    50     50    6,     60    60    7,    7,     8,    9,    10,   11,
  11,   12,    12,    13,    13,    3,    13,    3,   130    13,   13,   15,
  17,   190    130    130    150    10    10    1,   190    19,   19,   190
  190   19,    19,    19,    19,   190   190    16,   210   19,   13,   148
  140   140    160    3,     16,   218    58    190   13,   19,   13,   12,
   48    48     30    19,    19:   120   190    190   11,    80    8,    8,
   80    80     80    90     90    10,    10,                118    11,   12,
                                                15,    13,   13,   13,   13,
  13,   13,    13,    12,    12,   12o    16,    16,   13,    13,    5J
FILL G[*] WITH 00 13,   13,
   10    50     6,    60     3,    10    20    30    40    50    1,    5,
   50    60     60    60     7,    7,    7,    8,    90    13,   13,   10,
  118    11,    12,    128    138   138   138    14,   130   18,   18,   160
  178    17,    13,    13,    14,   19,    3,    19,   180                18,
   18,   180    18,    180    180   180    30    15,    1,   160   130   200
   40    200    140    15,    30    60    1,    148    3,    13,   3,    5,
   5,    13,     5,     3,    3,    40    50    60    7,    7,    7,    7,
   7,    70     7,     8,     80    10,    10,    11,   110   110   11,   12,
  138    13,    13,    13,    130   138   13,    13,   13,    13,   13,   13,
  130    138    13,    13,    13,   130   130   160   130   130    4J
```

```
FILL MTB[*] WITH 0,
    1,    2,    5,   16,   258  298   30,   338   390   42,   470  48,
  558   58,   6 2 : 68,   71,   75,   81,   84,  111,  122,  125,  136,
 1398 158, 1610  168 .  1710 174,183,  186, 198,201,204,  2160
 2238 2298 232, 235, 245, 256, 257, 258, 259, 2620 2650 268,
 271,274,277, 280, 283, 286, 289,  2908 291,292, 2930  297,
 3018 305, 3098  315, 320, 321, 324, 325, 328, 329, 332, 3330
 3370 3410 3428 347,  348, 3498 350, 351, 3528 3568 3570 358,
 3590 360, 361,  362,  3638 3648 368, 372, 3730 374, 375, 3740.
 3770 381, 385,  389, 3930 3978 401, 405, 4080 412, 4160 420,
  424, 428, 432, 4360 440, 4438 446r 454, 4558 458, 461)
FILL PRTB[*] WITH 0,
    0,-103,   98    0,   42,   57,-115,    38     44, 57,-116,  3,
 -117,   40    0,   68   57,-118,    4,    6,  67,-119,    20    0,
    7,-110,    7,    0,    0,-112,    68    0,  770'1010 11,-111,
    7,    0,-109,   80    0,   11,    9,-104,   90    0,   00   78,
   13,  -990  12,-108,    9,    0,-100,   12,    0, 138 -970    138
    0,   79,  -96,  140  -988  138    0, '950   150    0,  16,  -93,
   16, 0, 80, -92,  178  "940   168   -0, -90,   180    0,  -83,
   19,   82,  20,  -84,   19,   83,  200  -850  198  84, 20,  -860
  190   85,  20, -87, 19,   868   20,  -88,  19,  870  200  -89,
   19,    0,  88,  22,  -80,   21,   89,  220  -810  21, -82,  20,
    0,  -79,  21,    0,   90,   24,  -76,   23,  910   24, -77,   238
  -78,  220    08  -73,  238    08   92, 26, -68,  258  930   26,
  -69, 25,  940   26,  -70,   25,   950   26, -71, 25, -72,  248
   Or -67,  25,    0,   96,   28,  -65, 27,  -66,  26,    0 8 -64,
   27,    08 -46,  280    08 438  57,  -350  300   88  710  -37,
  298    08 "448  28,    08  8,   550  -31,  328  80   690   -33,
  310   69,  -340  310    08 '430  28,    0, -41,  280    0, -25,
  340  115,  360  -26,  348  1150 116, 36,  -270  340    0,  560
  360  -230  350  -24,  350   0,  38,  -210  370  "220  360    0,
  -20,  370    0, -400  280    08 -380  28,  310  -39,  280  740
    9,-105,   90  0,   640   80   65,   -5,  41,  56,  -6,   418
   -7,   400    08     0,   0, 0, -10,  38,   08  -11,  380    0,
  '120   38,   0,  -13,  38,   0,  -14,   380    0,  -15,  380    0,
 -160   38,   08 -17,  38,   0,        -180  38,    0,  -19,  380   0 8
   08    08   0,  0,   400  -30,  330   0 8  630  '18  420   0 8
  638   '20  438    0,   63,   -3,  440    0,  -4,. 410  58,-113,
   50    08  80   650  -48,  28,    0,   0,-114,  3 0    0,    0,
  -32,   32,   0,    0,  -36,  300    0,    0, 28,-106,   90    0,
    9,-107,    9,    0, 0,   80  76,-102,  10,    0,    0,    0,
   08    08    0, 19,  -91,  180      0,  0,   08    0,    0,    0,
    0,    0,  -0,  340  -74,  230  .0._.  240  -750  238    0,  0,
    0,  08  Or    0,   280  "580  28,    08  400  9590   28,  0,
  280  -600  280,        280  -61,  28,    08 280  -620   28,  0,
  280  -63,  280   On  280  -45, 28,    0, -490  280   0 8  40,
 -500  280    0,  408  -51,  280    0,  400  -52,  280    08  4 0 8
 -530  280    0,  400  -54,  280    0,  400  "558  280    0,  408
 '560   28,   0,  400  -57,  280    0, -42,  280    08 -47,  28,
    0,  36,  -280  34, 1168   36,  -29,  348      0, 0,  -8,   39,
    0,   -9,  390   0,   2, 119,-120,   1,  0)
WC + 8;  CC + 7;  CLEAR (WRITEBUFFER[0]); CLEAR (READBUFFER[0]);
S[0] + MARK; ERRORFLAG +    FALSE;
I+  J   +  BN  + ON + NP + PRP +0)
```

98

COMMENT ALGORITHM FOR SYNTACTIC ANALYSIS:
    COMPARE THE PRIORITIES OF THE SYMBOL R AN0 OF THE
    SYMBOL UN TOP OF THE STACK S. IF S[J]...S[I] CONSTITUTE A RIGHT-
    PART OF A PRODUCTION, THEN REPLACE THIS SEQUENCE BY THE
    CORRESPONDING LEFT-PART AND BRANCH TO THE INTERPRETATION-RULE
    BELONGING TO THE PERFORMED PRODUCTION;
A0:   R ← INSYMBOL;
A1:   IF F[S[I]] > G[R] THEN GO TO A2;
       IF R = MARK THEN GO TO A9;
       I ← J ← I+1; S[I] ← R; MOVE (NAME, V[I]); GO TO A0;
A21   IF F[S[J-1]] = G[S[J]] THEN BEGIN J ← J-1; GO TO A2 END;
       M ← MTB[S[J]];
A3:   IF PRTB[M] = 0 THEN BEGIN ERROR(5); GO TO EXIT END;
       N ← J;
A4:   N ← N+1;
       IF PRTB[M] < 0 THEN GO TO A8;
       IF N ≤ I THEN GO TO A7;
A5:   M. ← M+1;
       IF PRTB[M] ≥ 0 THEN GO TO A5;
A6:   M ← M+2; GO TO A3;
A7:   IF PRTB[M] ≠ SCN; THEN GO TO A5;
       M ← M+1; GO TO A4;
A8:   IF N 4 I THEN GO TO A61
       GO TO BRANCH[-PRTB[M]]; ·
LO: SC[J ← PRTB[M+1]; I ← J; GO TO A11'

COMMENT THE FOLLOWING ARE THE INTERPRETATION-RULES;
L1:
L2:   P1(S[J]); NP ← NP+1; MOVE (V[I],NL1[NP]); ZERO (V[I]);
       NL2[NP] ← BN; NL3[NP] ← ON ←ON+1; NL4[NP] ← S[J]; GO TO LO;
L3:   NP ← NP+1; MOVE (V[I],NL1[NP]); ZERO (V[I]);
       NL2[NP] ← BN; NL3[NP] ← NL4[NP] ← UNOEF; GO TO LO;
L4:   FOR T ← NP STEP -1 UNTIL 1 DO
       IF EQUAL (NL1[T], V[I]) THEN GO TO NAMEFOUND;
       ERROR (0); GO TO LO;
  NAMEFOUND:
       IF NL4[T] = NEWSYM THEN
          P3(REFSYM, NL3[T], NL2[T]) ELSE
       IF NL4[T] = LABSYM THEN
          P3(LABSYM, NL3[T], NL2[T]) ELSE
       If NL4[T] = FORSYM THEN
          BEGIN P3(REFSYM, NL3[T], NL2[T]); P1(VALSYM) END ELSE
          BEGIN P3(LABSYM, NL3[T], NL2[T]); NL3[T] ← PRP END ;
       GO TO LO;
L5:   P1(S[I]); GO TO LO;
L6:   P1(VALSYM); GO TO LO;
L10:
L9:   V[J] ← 0; GO TO LO;
L11:
L8:   V[J] ← 1; GO TO LO;
L12: V[J] ←2; GO TO LO;
L13:   V[J] ← 3; GO TO LO;
L14: V[J] ← 4; GO TO LO,
L15:   VC[J ← 5; GO TO LO;
L16: VC[J ← 61 GO TO LO;

99

```
L17:   V[J] ← 7; GO  TO  LO;
L18:  V[J] ← 8; GO  TO  LO;
L19:   V[J] ← 9; GO  TO  LO;
L20:   SCALE  ←1; GO TO LO;
L21:  V[J] ← V[J] × 10 +V[I]; SCALE  ← SCALE+1;
       IF SCALE  > 11 THEN   ERROR  (1); GO  TO  LO;
L23:  V[J] ← V[I] × IO *   ('SCALE)   ← V[J];GO  TO  LO;
L26:   V[J] ← V[J] × 10 ● V[I]; GO  TO  LO;
L27:  V[J] ← V[J] × .1+V[I]; GO  TO  LO;
L28:   V[J] ← 10 * V[I]; GO  TO  LO;
L29:   V[J] ← .1 * V[I]; GO  TO  LO;
L31:  V[J] ←V[J]+1; G  O TO  LO;
L32:   V[J] ← 0; GO  TO  LO;
L33:   P2(S[I], V[J]+1); GD  TO  LO;
L34:  P2(S[I], V[J]); GO  TO  LO;
L36:   BN ← BN+1; ON ← 0; P2(S[J], UNDEF); V[J]← PRP;
       NP ← NP+1; ZERO  (NL1[NP]); NL2[NP] ← MP; MP ← NP; GO  TO  LO;
L37:P1(S[I]); FIXUP  (V[J], PRP+1); NP ← MP-1; MP ← NL2[MP];
       BN ← BN-1;  GO  TO  LO;
L38: P1(VALSYM); GO  TO  LO,
L39:   P1(CALLSYM); GO  TO  LO;
L40:   P2(BOOLSYM,V[I]); G O TO  LO;
L41: P1(NUMSYM); PHP ← PRP+1; PROGRAM[PRP] ← V[I]; GO  TO  LO;
L42: P2(S[I], V[I]); GO  TO  LO;
L75: P1(UNARYMINUS); GO  TO  LO;
L92: L96:  L101;L102: P2(S[I], UNDEF); V[J] ← PRP; GO  TO  LO;
L93; L97: FIXUP  (V[J], PRP+1); GO  TO  LO;
L104:  FIXUP  (V[J], V[J+1]+1); FIXUP(V[J+1],PRP+1);GO T O LO;
L113: FOR T ← NP STEP -1 UNTIL MP+1 DO
       If EQUAL  (NL1[T],V[J])  THEN
       BEGIN IF NL4[T] ≠ UNDEF  THEN  ERROR(2);
          T1 ← NL3[T]; NL3[T] ← PRP+1; NL4[T] ← LABSYM; ZERO  (V[J]);
          L1131: IF T1 ≠ UNDEF  THEN
             BEGIN T ← PROGRAM[T1].BFIELD; FIXUP  (T1, PRP+1);
                   T1←T; GO  TO  L1131
             . END ; GO  TO  LO;
       END ;
       ERROR(3); GO  TO  LO;
L114:   BN  ← BN+1; ON ← 0; P1(S[I]);
       NP←NP+1; ZERO  (NL1[NP]); NL2[NP] ← MP; MP ← NP; GO  TO  LO;
L118:P1(S[I]);GO  TO  LO;
L119: FOR T ← MP+1 STEP 1 UNTIL NP  DO  If  NL4[T] = UNDEF  THEN  ERROR(4);
       NP←MP-1; MP ← NL2[MP];P1(S[I]); BN ← BN-1;  GO  TO  LO;

L45: L47:  L49; L50; L51; L52; L53; L54; L55; L56; L57; L58; L59; L60;
L61: L62:   L63:  L91; L106; L107;P1(S[J]);GOTOLO;
L65; L68;  L69:  L70; L71; L76; L77; L80; L81; L84; L85:  L86; L87; L88;
L89:  L99:  L105;P1(S[J+1]); GO TO LO)
L7:   L22:  L24; L25; L30; L35; L43; L44; L46; L48; L64; L66; L67; L72;
L73:  L74; L78; L79; L82; L83:  L90; L94; L95; L98:  L100; L103; L108;
L109; L110; L111;  L112; L115; L116; L117; L120; G  O TO LO;

A9:   P1(MARK); PROGRAMDUMP; If ERRORFLAG THEN GO  TO  EXIT
END ● ;
```

```
BEGIN  COMMENT  E U L E R  IV  INTERPRETER        MCKEEMAN  & WIRTH )
    REAL  ARRAY S, SI, F, FI [0:1022]]   COMMENT STACK)
    INTEGER  I1, I2, LVL, FORMALCOUNT)
    INTEGER  SP)       COMMENT  TOP-STACK POINTER)
    INTEGER  F Pi      COMMENT  FREE  STORAGE  SPACE  POINTER;
    INTEGER  MP)       COMMENT  BLOCK-  OR  PROCEDURE-MARK  POINTER;
    INTEGER  PP)       COMMENT  PROGRAM  POINTER)
    LABEL  ADD, SUB, MUL, DIVIDE,  IDIV, REMAINDER,  POWER, NEG, ABSV,
    INTEGERIZE, REALL, LOGICAL, MIN, MAX,  EQL, NEQ, LSS, LEQ, GEQ, GTR,
    LENGTH,  ISLOGICAL, ISNUMBER, ISREFERENCE,  ISLABEL, ISSYMBOL,
    ISLIST, ISPROCEDURE, ISUNDEFINED, LAND, LOR, LNOT, LEFTQUOTE,
    RIGHTQUOTE,  RIGHTPAREN, REFERENCE,  PROCEDURECALL,  VALUEOPERATOR,
    GOTO, NEW, FORMAL,  BEGINV, ENDV, STORE, THENV, ELSEV, NUMBER, LOGVAL,
    LABELL, SUBSCRIPT, SEMCOLON, UNDEFIND, OUTPUT, INPUT, TAIL,
    CATENATE,  LISTT, SYMBOL, DONE, UNDEFINEDOPERATOR, NEXT,  TRANSFER;
```

COMMENT   SI AND FI FIELD DEFINITIONS

		1-4	8-17	18-27	28-37	38-47	48-97
NUMBER	TYPE						VALUE
BOOLEAN	TYPE						VALUE
SYMBOL	TYPE						VALUE
UNDEFINED	TYPE						
LIST	TYPE				LENGTH	ADDRESS	
REFERENCE	TYPE				MARK	ADDRESS	
LABEL	TYPE				MARK	ADDRESS	
PROCEDURE	TYPE			BLOCK NO.	MARK	ADDRESS	
BLOCKMARK	TYPE		DYNAMIC	BLOCK NO.	STATIC	ADDRESS	LIST)

```
- DEFINE
TYPE=[1:4]#,
WCT=[28:10]#,
ADDRESS=[38:10]#,
STATIC=[28:10]#,
DYNAMIC=[8:10]#,
BLN=[18:10]#,
NSA=[18:10]#,           COMMENT  NEW  STARTING  ADDRESS  FOR  FREE)

UNDEFINED=0#,
NUMBERTYPE=1#,
SYMBOLTYPE=2#,
BOOLEANTYPE=3#,
LABELTYPE=4#,
REFERENCETYPE=5#,
PROCEDURETYPE=6#,
LISTTYPE=7#,
BLOCKMARK=8# )

STREAM  PROCEDURE  MOVE(F1, T1, W))
BEGIN  LOCAL  R1, R2)
    SI + W)   SI + SI + 6)
    DI + LOC R1)   DI + DI + 7)   DS + CHR)
    DI + LOC R2)   DI + 01  + 7)  DS + CHR)
    SI + F1)  DI + T1)
    R1(2(DS + 32 WDS)))  DS + R2 WDS)
END)
```

```
PROCEDURE DUMPOUT(XI, X);   VALUE XI, X; REAL XI, X;
BEGIN INTEGER T, Ii

PROCEDURE LISTOUT(XI); VALUE XI;   REAL XI;
BEGIN COMMENT RECURSIVE  LIST  OUTPUT;
    INTEGER I, N;
    SWITCH FORMAT LPAR +
        ("("),(",("),(",,("),(",,,("),(",,,,("),(",,,,,("),(",,,,,,(");
    SWITCH FORMAT RPAR +
        (")"),(",)"),(",,)"),(",,,)"),(",,,,)"),(",,,,,)"),(",,,,,,)");
    WRITE(<X9,"LIST",I10>, XI.ADDRESS);  WRITE ((NO), LPAR[LVL]);
    LVL + LVL + 1;  N + XI.ADDRESS + XI.WCT -1;
    FOR I + XI.ADDRESS STEP 1 UNTIL N DO DUMPOUT (F[I], F[I]);
    LVL + LVL - 1; WRITE (RPAR[LVL]);
END LIST OUT;

    T + XI.TYPE;
    IF T = UNDEFINED THEN WRITE(<X9, "UNDEFINED">) ELSE
    If T = NUMBERTYPE THEN
    BEGIN
        If X # ENTIER(X) THEN WRITE(<X9,"NUMBER",E20,10>, X) ELSE
        WRITE(<X9, "NUMBER"0 I20>, X)
    END ELSE
    IF T = BOOLEANTYPE THEN WRITE(<X9,"LOGICAL", 14X1, L5>, BOOLEAN(X))
    ELSE
    IF T = LISTTYPE THEN LISTOUT(XI) ELSE
    IF T = LABELTYPE THEN WRITE(<X9, "LABEL0 ADDRESS =", I4,
    " MARK""0 I4>, XI.ADDRESS, XI.STATIC) ELSE
    If T = REFERENCETYPE THEN WRITE(<X9,"REFERENCE, ADDRESS=",I4,
    " MARK=",I4>,XI.ADDRESS,XI.STATIC) ELSE
    If T = PROCEDURETYPE THEN
    WRITE(<X9,"PROCEDURE DESCRIPTOR0 ADDRESS=", I4, " BN=", I4,
    " MARK""0 I4>, XI.ADDRESS, XI.BLN, XI.STATIC) ELSE
    IF T = BLOCKMARK THEN
        WRITE(<X9, "BLOCKMARK, BN=", I4, " DYNAMIC""0 140 " STATIC=",
        I4, " RETURN""0 I4>, XI.BLN, XI.DYNAMIC,XI.STATIC,XI.ADDRESS)
    ELSE IF T = SYMBOLTYPE THEN
    WRITE(<X9, "SYMBOL     ",A5>, X));
END DUMPOUT;

PROCEDURE ERROR(N);  VALUE N;  INTEGER N;
BEGIN INTEGER I;
    SWITCH FORMAT ER +
    ("ILLEGAL INSTRUCTION ENCOUNTERED"10
    ("IMPROPER OPERAND TYPE"),
    ("CANNOT DIVIDE BY 0"),
    ("CALL OPERATOR DID NOT FIND A PROCEDURE")0
    ("REFERENCE OR LABEL OUT OF SCOPE"),
    ("OUT Of SCOPE ASSIGNMENT OF A LABEL OR A REFERENCE")0
    ("SUBSCRIPT IS NOT A NUMBER"),
    ("SUBSCRIPT NOT APPLIED TO A VARIABLE")0
    ("SUBSCRIPTED VARIABLE IS NOT A LIST"),
    ("SUBSCRIPT IS OUT OF BOUNDS"),
    ("CANNOT TAKE TAIL OF A NULL LIST"),
    ("STACK OVERFLOW"),
```

```
          ("STACK  OVERFLOW DURING  GARBAGE  COLLECTION"),
          ("ASSIGNMENT  TO  A NON-VARIABLE   ATTEMPTED"),
          ("FREE  STUKAGE  AREA  IS  TOO  SMALL");
          WRITE  ([OBL], ER[N]);
          WRITE  (</ "SP=",I4," FP=",I4," PP=",I4," MP=",I4," SYL=",I4/>,
                  SP, FP, PP, MP, PROGRAM[PP].AFIELD));
          FOR I + 1 STEP 1 UNTIL Sip DO
          BEGIN  WRITE([NO], <I4>, I)) DUMPOUT (SI[I],S[I])  END )
          GO TO DONE
END ERROR;

PROCEDURE  FREE(NEED)) VALUE NEED) INTEGER NEED;
      COMMENT  "FREE"  IS  A "GARBAGE  COLLECTION"  PROCEDURE.   IT IS  CALLED
          WHEN FREE STORAGE f IS USED UP, AND MORE SPACE IS NEEDED.
          GARBAGE  COLLECTION  TAKES  THE  FOLLOWING  STEPS:
          1.   ALL  BLOCKMARKS,  LIST  DESCRIPTORS  AND  REFERENCES  IN  STACK
          POINT  TO  VALID  INFORMATION  IN  FREE  STORAGE.    LIKEWISE,  ALL
          LIST DESCRIPTORS AND  REFERENCES  THAT  ARE  POINTED  TO  ARE  VALID,
          ENTER  INTO  THE  STACK  ALL  SUCH  ENTITIES,
          2.   THE  GARBAGE  COLLECTOR MUST  KNOW IN  WHICH  ORDER  TO  COLLAPSE  THE
          FREE  STORAGE.   THUS SORT  THE  LIST  BY  FREE  STORAGE  ADDRESS,
          3.   MOVE  EACH  BLOCK  DOWN  If   NECESSARY,
          4.   NOW  THE  ADDRESSES  ARE  WRONG--MAKE  ONE  MORE  PASS  THROUGH  THE
          SORTED  LIST  TO  UPDATE  ALL  ADDRESSES;
BEGIN  OWN  INTEGER  G, H, I, J)   OWN  REAL  T)

INTEGER  PROCEDURE  FIND(W)) VALUE W,   REAL W)
BEGIN  COMMENT  BINARY  SEARCH  THROUGH  ORDERED TABLE)
      INTEGER  T, N, B, KEY,  K)
      LABEL  FOUND, BINARY)
      T + G+1)  B + SP + 1;
      KEY + W.ADDRESS)
      BINARY:    N • (B+T) DIV 2;
      K + SI[N].ADDRESS)
      If  K = KEY THEN GO TO FOUND;
      IF K < KEY THEN B + N ELSE T + N)
      GO TO BINARY;
      FOUND:   FIND + SI[N].NSA
END FIND;

PROCEDURE  RESET(W, Z))   REAL W, Z)
BEGIN  INTEGER  TY)
      TY + W.TYPE)
      If  TY = REFERENCETYPE OR TY = LISTTYPE THEN W.ADDRESS + FIND(W) ELSE
      IF TY = BLOCKMARK  THEN Z.ADDRESS + FIND(Z)
END RESET;

PROCEDURE  VALIDATE(P);   VALUE  P) REAL P)
BEGIN  COMMENT  TREE  SEARCH  FOR  ACTIVE  LIST  STORAGE;
      INTEGER  I, U;
      G + G + 1;
      IF  G > 1022  THEN  ERROR(12))
      SI[G] + P)
      U + P.ADDRESS + P.WCT -1)
      IF P.TYPE = LISTTYPE  THEN  FOR  I+ P.ADDRESS  STEP  1  UNTIL  U  DO
```

```
    IF F[I].TYPE = LISTTYPE OR F[I].TYPE = REFERENCETYPE    THEN
    VALIDATE(F[I]))
END VALIDATION)

PROCEDURE   SORT(LB, UB))  VALUE  LB, UB)   INTEGER  LB, UB)
BEGIN  COMMENT  BINARY  SORT)
    INTEGER M)
PROCEDURE   MERGE(LB, M, UB))   VALUE  LB, M, UB)    INTEGER   LB, M, UB)
BEGIN  INTEGER  K,L,U,K1, K2)   LABEL  A, B)
    K + UB - LB)
    MOVE(SI[LB], S[LB],  K))
    L + K + LB)    U + M) GO TO B)
A:  K1 + S[L].ADDRESS)   K2 + S[U].ADDRESS)
    IF  K1  < K2  UR (K1 = K2  AND  S[L].TYPE = LISTTYPE)   THEN
    BEGIN  SI[K]+S[L])L+ L+1
    END ELSE
    BEGIN  SI[K]+ S[U]) U + U+1
    END)
    K + K + 1)
B: IF  L = M  THEN ELSE  IF U = UB  THEN
    BEGIN  K  + M-L) MOVE(S[L],SI[UB-K], K)
    END ELSE GO TO A
END MERGE)

    IF  LB < UB  THEN
    BEGIN  M + (LB+UB) DIV 2)
        SORT(LB, M)) SORT(M+1,  UB)) MERGE(LB,  M+1,  UB+1)
_   END
END  SORT)

    INTEGER  LLA, LLW)
    G + SP)
    FOR  H+1  STEP 1  UNTIL  SP  DO
    BEGIN  CUMMENT  LOCATE  ALL  ACTIVE LISTS AND REFERENCES)
        IF S[H].TYPE = LISTTYPE OR S[H].TYPE . REFERENCETYPE  THEN
        VALIDATE(SI[H]) ELSE
        IF  S[H].TYPE = BLOCKMARK  THEN VALIDATE(S[H]))
    END)
    COMMENT  SORT  THEM IN  ORDER  OF  INCREASING ADDRESS)
    SORT(SP+1, G))
    I t 1) COMMENT  COLLAPSE  THE  FREE STORAGE)
    FOR J  + SP + 1 STEP 1 UNTIL G DO
    IF  SI[J].TYPE . LISTTYPE  THEN
    BEGIN  CUMMENT IF G.C.  OCCURS  DURING  "COPY"  THEN  WE  MUST AVOID
        THE CREATION OF DOUBLE LIST ENTRIES FROM DUPLICATED DESCRIPTORS)
        IF SI[J]=SI[J+1] THEN SI[J+1].TYPE t  UNDEFINED)
        LLA  + SI[J].ADDRESS)  LLW . SI[J].WCT)
        IF  LLA  ≠ I THEN
        BEGIN
            MOVE(F[LLA], F[I], LLW))
            MOVE(FI[LLA],FI[I], LLW))
        END)
        SI[J].NSA + Ii
        I + I + LLW)
    END  ELSE  SI[J].NSA +I= LLW + SI[J].ADDRESS - LLA)
```

104

```
        FP ← I;

        COMMENT  RESET  ALL  AFFECTED  ADDRESSES)
        FOR I ← 1  STEP  1  UNTIL  SP DO RESET(SI[I],S[I]);
        FOR I ← 1  STEP  1  UNTIL  FP-1 DO RESET(FI[I],F[I]);
        IF FP ← NEED > 1022 THEN ERROR(14);
END  FREE ;

PROCEDURE  MOVESEG(LD); REAL  LD;
BEGIN  COMMENT  MOVE  ONE  LIST  SEGMENT;
        INTEGER  W, X;
        W ← LD.WCT;
        IF FP ← W > 1022  THEN FREE(W);
        X ← LD.ADDRESS;
        MOVE(F[X], F[FP], W);
        MOVE(FI[X],FI[FP], W);
        LD.ADDRESS ← FP;
        FP ← FP ← W;
END  MOVE  SEGMENT;

PROCEDURE  COPY(LD); REAL  LD;
BEGIN  INTEGER I,  J;              COMMENT   RECURSIVE   LIST COPY;
        MOVESEG(LD);
        J ← LD.WCT . 1;
        FOR I ← 0  STEP 1 UNTIL  J  DO
        IF FI[I+LD.ADDRESS].TYPE = LISTTYPE THEN COPY(FI[I+LD.ADDRESS])
END  COPY;

-  PROCEOURE  BOOLTEST; IF SI[SP].TYPE ≠ BOOLEANTYPE  THEN  ERROR(1);

INTEGER PROCEDURE  ROUND(X); VALUE  X; REAL  X; ROUND ← X;

PROCEDURE  BARITH;
BEGIN  If  SI[SP].TYPE ≠ NUMBERTYPE OR SI[SP-1].TYPE ≠ NUMBERTYPE THEN
        ERROR(1) ELSE  SP ←SP-1
END  BARITH;

PROCEDURE  FETCH;
BEGIN  INTEGER  Ii
    IF SI[SP].TYPE = REFERENCETYPE  THEN
        BEGIN I ← SI[SP].ADDRESS; SI[SP] ← FI[I]; S[SP]← F[I]  END
END  FETCH ;

INTEGER  PROCEDURE  MARKINDEX(BL); VALUE  BL; INTEGER BL;
BEGIN COMMENT MARKINDEX IS THE  INDEX OF  THE  MARK WITH BLOCKNUMBER BL;
    LABEL U1; INTEGER I;
    I ← MP;
    U1: IF SI[I].BLN > BL   THEN
    BEGIN I ←SI[I].STATIC; GO TO  U1 END;
    If  SI[I].BLN < BL THEN ERROR(4);
    MARKINDEX ← I
END  MARKINDEX ;

PROCEDURE LEVELCHECK(X, Y); VALUE  Y; INTEGER  Y; REAL  X;
BEGIN    INTEGER   T, 18 t8 U; T ← X.TYPE;
```

105

```
    If  T =  REFERENCETYPE  OR  T = LABELTYPE  THEN
        BEGIN IF X.STATIC > Y THEN ERROR(5)  END  ELSE
    IF  T = PROCEDURETYPE  THEN  X.STATIC ← Y  ELSE
    IF  T = LISTTYPE  THEM
    BEGIN  L ← X.ADDRESSJ  U ← L ← X.WCT -1J
        FOR I ← L  STEP  1  UNTIL  U  DO  LEVELCHECK(FI[I],Y)
    END
END  LEVEL  CHECKJ

PROCEOURE  SPUPJ  IF  SP ≥ 1022  THEN  ERROR(11) ELSE SP← SP + 1J

PRDCEOURE  SETIS(V)J  VALUE  VJ  INTEGER  VJ
BEGIN FETCHJ
    S[SP] ← REAL(SI[SP].TYPE = V)J
    SI[SP].TYPE ← BUOLEANTYPEJ
END  SET  ISJ

SWITCH EXECUTE ←
    PROCEOURECALL,  VALUEOPERATOR  SEMICOLON, UNDEFINEOOPERATOR,
    REFERENCE8  NEW8  FORMAL, LABELL, UNDEFINEDOPERATOR, LOGVAL,
    SUBSCRIPI, BEGINV, ENDV, NUMBER, RIGHTPAREN, LEFTQUOTE, RIGHTQUOTE,
    GOTO8  OUTPUT8  STORE, UNDEFINEDOPERATOR, THENV, ELSEV, CATENATE8
    LOR, LAND, LNOT, EQL, NEQ,LSS,LEQ, GEQ, GTR,MIN, MAX8
    ADD8  SUB, MUL, UIVIOE, IDIV, REMAINDER, POWER, ABSV,LENGTH,
    INTEGERIZE, REALL,LOGICAL,LISTT, T A I L ; INPUT,
    ISLOGICAL, ISNUMBER, ISREFERENCE, ISLABEL,ISLIST,ISSYMBOL,
    ISPROCEDURE  ISUNDEFINED, SYMBOL8  UNDEFIND, UNDEFINEOOPERATOR, NEG,
    UNDEFINEOOPERATOR, UNDEFINEDOPERATOR, DONEJ

    WRITE ([PAGE])J
    SP←MP← PP← OJ  FP ← 1J  LVL ← OJ  FT ← FT+9J

NEXTI  PP  ← PP+1J
TRANSFERI  GO To  EXECUTE [PROGRAM[PP].AFIELD - FT]J

UNDEFINEOOPERATORI
    ERROR(0)J
SEM COLON:
    se ← SP - 1J GO TO NEXTJ
UNDEFIND:    SPUPJ
    SI[SP].TYPE ← UNDEFINEDJ GO TO NEXTJ
NUMBER:
    PP ← PP + 1J SPUPJ
    SI[SP].TYPE ← NUMBERTYPEJ  SCSPJ ← PROGRAM[PP]J GO TO NEXTJ
SYMBOLI  SPUPJ
    SI[SP].TYPE ← SYMBOLTYPEJ  SCSPJ ← PROGRAM[PP].BFIELOJ GO TO NEXTJ
LOGVALI  SPUPJ
    SI[SP].TYPE ← BOOLEANTYPEJ  SCSPJ ← PROGRAM[PP].BFIELOJ
    GO TO NEXTJ
REFERENCE:    SPUPJ
    SI[SP] ← OJ
    SI[SP].TYPE ← REFERENCETYPEJ
    SI[SP].STATIC ← 11 ← MARKINDEX(PROGRAM[PP].CFIELO)J
    SI[SP].ADDRESS ← S[I1].ADDRESS ← PROGRAM[PP].BFIELD . 1J
    GO TO NEXTJ
```

106

```
LABELL:   SPUP;
    SI[SP].TYPE ← LABELTYPE;
    SI[SP].STATIC ← MARKINDEX(PROGRAM[PP].CFIELD);
    SI[SP].ADDRESS ← PROGRAM[PP].BFIELD; GO TO NEXT;
CATENATE:
    IF SI[SP].TYPE ≠ LISTTYPE OR SI[SP-1].TYPE≠LISTTYPE THEN ERROR(1);
    IF SI[SP-1].ADDRESS+SI[SP-1].WCT ≠ SI[SP].ADDRESS  THEN
    BEGIN COMMENT MUST HAVE CONTIGUOUS LISTS;
        MOVESEG(SI[SP-1]);
        MOVESEG(SI[SP]);
    END;
    SP ← SP - 1;
    SI[SP].WCT ← SI[SP].WCT + SI[SP+1].WCT;
    GO TO NEXT)
LOR:    BOOLTEST;
    IF NOT BOOLEAN(S[SP]) THEN BEGIN SP ← SP - 1; GO TO NEXT END;
    PP ← PROGRAM[PP].BFIELD; GO TO TRANSFER;
LAND:   BOOLTEST;
    IF BOOLEAN(S[SP]) THEN BEGIN SP ← SP - 1; GO TO NEXT END;
    PP ← PROGRAM[PP].BFIELD; GO TO TRANSFER;
LNOT:   BOOLTEST;
    S[SP] ← REAL(NOT BOOLEAN(S[SP])); GO TO   NEXT;
LSS:    BARITH;
    S[SP] ← REAL(S[SP] < S[SP+1]);
    SI[SP].TYPE ← BOOLEANTYPE; GO TO NEXT;
LEQ:    BARITH;
    S[SP] ← REAL(S[SP] ≤ S[SP+1]);
    SI[SP].TYPE ← BOOLEANTYPE; GO TO NEXT;
EQL:    BARITH;
    S[SP] ← REAL(S[SP] = S[SP+1]);
    SI[SP].TYPE ← BOOLEANTYPE; GO TO NEXT;
NEQ:    BARITH;
    S[SP] ← REAL(S[SP] ≠ S[SP+1]);
    SI[SP].TYPE ← BOOLEANTYPE; GO TO NEXT;
GEQ:    BARITH;
    S[SP] ← REAL(S[SP] ≥ S[SP+1]);
    SI[SP].TYPE ← BOOLEANTYPE; GO TO NEXT;
GTR:    BARITH;
    S[SP] ← REAL(S[SP] > S[SP+1]);
    SI[SP].TYPE ← BOOLEANTYPE; GO TO NEXT;
MN:     BARITH;
    IF S[SP+1] < S[SP] THEN S[SP] ← S[SP+1]; GO TO NEXT;
MAX:    BARITH;
    IF S[SP+1] > S[SP] THEN S[SP] ← S[SP+1]; GO TO NEXT;
ADD:    BARITH;
    S[SP] ← S[SP]+S[SP+1]; GO TO NEXT;
SUB:    BARITH;
    S[SP] ← S[SP] - S[SP+1]; GO TO NEXT;
NEG:    IF SI[SP].TYPE ≠ NUMBERTYPE THEN ERROR(1);
    S[SP] ← - S[SP]; GO TO NEXT;
MUL:    BARITH;
    S[SP] ← S[SP]×S[SP+1]; GO TO NEXT;
DIVIDE: BARITH;
    IF S[SP+1] = 0 THEN ERROR(2);
    S[SP] ← S[SP] / S[SP+1]; GO TO   NEXT;
```

```
IDIV:   BARITH;
    IF ROUND(S[SP+1]) = 0 THEN ERROR(2);
    S[SP]+ROUND(S[SP])DIVROUND(S[SP+1]);GO T O  N E X T ;
REMAINDER:   BARITH;
    IF S[SP+1] = 0  THEN ERROR(2);
    S[SP] + S[SP] MOD S[SP+1];  GO  TO  NEXT;
POWER:   BARITH;
    S[SP] + S[SP] * S[SP+1];  GO  TO  NEXTJ
ABSV :   IF  SI[SP].TYPE≠NUMBERTYPE  THEN  ERROR(1);
    S[SP] + ABS(S[SP]);  GO  TO  NEXT;
INTEGERIZE:
    IF  SI[SP].TYPE > BOOLEANTYPE  THEN  ERROR(1);
    S[SP] + ROUND(S[SP]);GO T O  NEXT;
REALL:
    IF  SI[SP].TYPE > BOOLEANTYPE  THEN  ERROR(1);
    SI[SP].TYPE + NUMBERTYPE; GO TO NEXT;
LOGICALl:
    IF  SI[SP].TYPE ≠ NUMBERTYPE  THEN  ERROR(1);
    IF  S[SP] = 0 OR  S[SP] = 1  THEN  SI[SP].TYPE+ BOOLEANTYPE  ELSE
    SI[SP].TYPE+   UNDEFINEO;
    GO TO NEXT;
LISTT:
    IF  SI[SP].TYPE ≠  NUMBERTYPE  THEN  ERROR(1);
    1 2 + S[SP];
    IF 12 + FP > 1022  THEN  FREE(I2);
    FOR 11 +FP STEP  1  UNTIL FP+I2-1 D O  FI[I1].TYPE + UNDEFINED;
    S I C S P J r T Y P E + LISTTYPE; SI[SP].WCT + 12; SI[SP].ADDRESS+FP;
    FP+  FP  +I2;  GO  TO NEXT;

ISLOGICAL: SETIS(BOOLEANTYPE);  GO  TO  NEXTJ
ISNUMBER:   SETIS(NUMBERTYPE); GO TO NEXT;
ISREFERENCE:   SETIS(REFERENCETYPE);  GO  TO  NEXT;
ISLABEL:   SETIS(LABELTYPE);  GO TO NEXTJ
ISLIST:   SETIS(LISTTYPE); GO T O  NEXT;
ISSYMBOL:   SETIS(SYMBOLTYPE);  GO  TO  NEXT;
ISPROCEDURE:   SETIS(PROCEDURETYPE);  GO  TO  NEXT;
ISUNDEFINED:   SETIS(UNDEFINED);  GO  TO  NEXTJ

TAIL:
    IF  SI[SP].TYPE≠LISTTYPE  THEN  ERROR(1);
    IF SI[SP].WCT = 0  THEN  ERROR(10);
    SI[SP].WCT  + SI[SP].WCT - 1;
    SI[SP].ADDRESS + SI[SP].ADDRESS + 1;  GO  TO  NEXT;
THENV:
    BOOLTEST;  SP  +SP-1;
    IF  BOOLEAN(S[SP+1]) THEN  GO  TO NEXT;
    PP + PROGRAM[PP].BFIELD;  GO  TO  TRANSFER;
ELSEV:
    PP + PROGRAM[PP].BFIELD;  GO  TO  TRANSFER;
LENGTH:
    FETCH;
    IF  SI[SP].TYPE ≠ LISTTYPE  THEN  ERROR(1);
    SI[SP].TYPE + NUMBERTYPE; S[SP] + SI[SP].WCT;  GO  TO  NEXT;
GOTO :
    IF SI[SP].TYPE ≠ LABELTYPE  THEN  ERROR  ( 1 );
```

108

```
    MP  ←SI[SP].STATIC;
    COMMENT WE MUST RETURN TO THE BLOCK WHERE THE LABEL IS DEFINED;
    PP ← SI[SP].ADDRESS; SP ← MP; GO TO TRANSFER;
FORMAL:
    FORMALCOUNT   ←FORMALCOUNT+1;
    IF FORMALCOUNT≤S[MP].WCT THEN  GO TO NEXT ELSE GO TO NEW;
NEW:
    S[MP].WCI ← S[MP].WCT +1;
    FI[FP].TYPE ← UNDEFINED;
    FP ← FP +1;
    IF FP > 1022 THEN FREE(1);
    GO TO NEXT;
STORE :
    IF SI[SP-1].TYPE ≠ REFERENCETYPE THEN ERROR(13);
    LEVELCHECK(SI[SP], SI[SP-1].STATIC);
    SP ← SP -1;                              COMMENT NON-DESTRUCTIVE STORE;
    I1 ← SI[SP].ADDRESS;
    S[SP] ← F[I1] ← S[SP+1];   SI[SP] ← FI[I1] ← SI[SP+1];
    COMMENT THE NON-DESTRUCTIVE STORE IS NOT APPLICABLE TO LISTS;
    IF SI[SP].TYPE=LISTTYPE THEN SI[SP].TYPE←UNDEFINED;
    GO TO NEXT;
SUBSCRIPT:
    If SI[SP].TYPE ≠ NUMBERTYPE THEN ERROR(6);
    SP ← SP -1;
    IF SI[SP].TYPE ≠ REFERENCETYPE THEN ERROR(7);
    I I ← SI[SP].STATIC; SI[SP] ← FI[SI[SP].ADDRESS];
    IF SI[SP].TYPE ≠ LISTTYPE THEN  ERROR(B);
    IF S[SP +1] < 1 Of ? S[SP+1] > SI[SP].WCT  THEN  ERROR(9);
    SI[SP].ADDRESS ← SI[SP].ADDRESS +S[SP+1] . 1;
    SI[SP].TYPE t REFERENCETYPE;        COMMENT MUST CREATE A REFERENCE;
    SI[SP].STATIC←I1; GO TO NEXT;
BEGINV: SPUP;
    SI[SP] ← O ;
    SI[SP].TYPE ← BLOCKMARK;
    SI[SP].BLN ← SI[MP].BLN + 1;
    SI[SP].DYNAMIC ← MP;
    SI[SP].STATIC ← MP;
    S[SP].TYPE ← LISTTYPE;
    S[SP].ADDRESS ← FP;
    S[SP].WCT ← O ;            COMMENT A NULL LIST;
    MP ← SP; GO TO NEXT;
ENDV:
    11        ← SI[MP].DYNAMIC;   .  -
    LEVELCHECK(SI[SP], SI[MP].STATIC);
    SI[MP]← SI CSP;; S[MP] ← S[SP];
    SP ← MP; MP ←I1; GO TO NEXT;
LEF TQUOTE :                        COMMENT PROCEDURE DECLARATION;
    SPUP;
    SI[SP].TYPE ← PROCEDURETYPE;
    SI[SP].ADDRESS← P P ;
    COMMENT THE PROCEDURE DESCRIPTOR MUST SAVE ITS OWN LEXICOGRAPHICAL
    LEVEL AS WELL AS THE STACK MARKER FOR UPLEVEL ADDRESSED VARIABLES;
    SI[SP].BLN ← SI[MP].BLN + 1;
    SI[SP].STATIC ← MP;
    PP ← PROGRAM[PP].BFIELD; GO TO TRANSFER)
```

109

```
RIGHTQUOTE:
    PP • SI[MP].ADDRESS;                          COMMENT  A  PROCEDURE  RETURN;
    1 1 ← SI[MP].DYNAMIC;
    LEVELCHECK(SI[SP], SI[MP].STATIC);
    SI[MP] ← SI[SP]; S[MP] ← SCSPJJ
    SP ← MP; MP ←11; GO  TO  NEXT;
VALUEOPERATOR:
    IF SI[SP].TYPE = LISTTYPE THEN  GO  TO  NEXT)
     FETCH;
    IF SI[SP].TYPE = PROCEDURETYPE  THEN
    BEGIN FORMALCOUNT ← 0;
        1 1 ← SI[SP].ADDRESS;
        SI[SP].TYPE ← BLOCKMARK;
        SI[SP].ADDRESS ← PP;
        SI[SP].DYNAMIC ← MPJ
        S[SP].TYPE ← LISTTYPE;
       .S[SP].WCT ← 0  ;
        MP   ← SP; PP ←11;
    END ELSE  IFSI[SP].TYPE=LISTTYPE  - THEN  COPY(SI[SP]);
    GO TO NEXT;
PROCEOURECALL:
    SP ← SP  -1; FETCH;
    If SI[SP].TYPE ≠ PROCEDURETYPE  THEN  ERROR(3);
    FORMALCOUNT  ← 0;
    1 1 ← SI[SP].ADDRESS;
    SI[SP].TYPE ← BLOCKMARK;
    SI[SP].ADDRESS ← PPJ
    SI[SP].DYNAMIC← MPJ
    S[SP] ← SI[SP+1];             COMMENT  THE  LIST DESC. FOR  PARAMETERS;
    MP ← SPJ PP • 11; GO TO NEXT;
RIGHTPAREN:
    11 ← PROGRAM[PP].BFIELD;
    If  11 ←FP > 1022 THEN  FREE(11);
    SP ← SP  - 11 ← 1;
    MOVE(S[SP], F[FP], 11); MOVE(SI[SP], FI[FP], 11);
    SI[SP].TYPE ← LISTTYPE;
    SI[SP].WCT ← 11;
    SI[SP].ADDRESS ← FPJ
    FP ← FP ←11; GU TO NEXT;
INPUT:    SPUPJ
    READ(S[SP])[EXIT]; SI[SP].TYPE← NUMBERTYPE; GO  TO  NEXT;
OUTPUT :
    DUMPOUT(SI[SP],S[SP]); 'GO TO  NEXT;
DONE :
END  INTERPRETER;

EXIT :
END .
```

```
001              BEGIN  NEW FOR; NEW MAKE; NEW T; NEW A;
006              FOR + LO FORMAL CV; FORMAL  LB;  FORMAL STEP;   FORMAL UB; FORMAL S;
013              BEGIN
013                  LABEL  L; LABEL  K;
014                  CV + LB;
020                  K;  If CV S UB THEN S ELSE: GOTO L;
036                  CV + CV + STEP;
047                  GOTO KC
051                  L; 0
052              END RQ;
057
057              MAKE + LQ FORMAL B; FORMAL X;
062              BEGIN  N&W T; NEW It  NEW F; NEW L;
067                  4 + B; T + LIST  L[1];
081                  F + IF LENGTH L # 1 THEN MAKE(TAIL L; X) ELSE XI
103                  FOR(@I; 1; 1; L[1]; LQ T[I] + F RQ) J
126                  T
127              END RQ;
132
132                  A +();
136                  FOR (@T; 1; 1; 4; LO BEGIN A + A & (T); OUT MAKE(@A;T)END RQ)
165              END S
```

PROGRAM DUMP

001	BEGIN				030	•		
002	NEW				031	•		
003	NEW				032.	ELSE	036	
004	NEW				033	LABEL	052	003
00s	NEW				034	•		
006	•	001	001		035	GOTO		
007	LQ	056			036	;		
008	FORMAL				037	•	001	002
009	FORMAL				038	•		
010	FORMAL				039	•	001	002
011	FORMAL				040	•		
012	FORMA4				041	•		
013	BEGI N				042	•	003	002
014	•	001	002		043	•		
015					044	•		
016	•	002	002		045	+		
017	•				046	+		
018	•				047	;		
019	+				048	LABEL	021	003
020	;				049	•		
021	•	001	002		050	GOTO		
022	•				051	;		
023	•				052	(	0,00000000•+00	
024	•	004	002		054	END		
025	•				055	RQ		
026	•				056	+		
027	S				057	;		
028	THEN	033			058	•	ooa	001
029	•	005	002		059	LQ	131	

111

Line	Op	Col1	Col2
060	FORMAL		
061	FORMAL		
062	BEGIN		
063	NEW		
064	NEW		
065	NEW		
066	NEW		
067	Q	004	003
068	Q	001	002
069	.		
070	.		
071	+		
072	)		
073	→	001	003
074	→	004	003
075	(	1.00000000e+00	
077	)		
078	.		
079	LIST		
080	+		
081	)		
082	→	003	003 ,
083	→	004	003
084	(symbols)		
083	(	1.00000000e+00	
087	≠		
088	(symbols)	099	
089	P	002	001
090	P	004	003
091	.		
092	TAIL		
093	Q	002	002
094	.		
095	.		
096	)	002	
097	,		
098	ELSE	102	
099	Q	002	002
100	.		
101	.		
102	+		
103	)		
104	Q	001	001
105	Q	002	003
106	(	1.00000000e+00	
108	(	1.00000000e+00	
110	Q	004	003
111	(	1.00000000e+00	
113	)		
114	.		
115	LQ	124	
116	P	001	003
117	Q	002	003
118	.		
119	)		
120	Q	003	003
121	.		
122	+		
123	RQ		
124	)	005	
125	,		
126	)		
127	Q	001	003
128	.		
129	END		
130	RQ		
131	+		
132	)		
133	Q	004	001
134	)		
135	+		
136	)		
137	Q	001	001
138	Q	003	001
139	(	1.00000000e+00	
141	(	1.00000000e+00	
143	(	4.00000000e+00	
145	LQ	165	
146	BEGIN		
147	Q	004	001
148	Q	004	001
149	.		
150	P	003	001
151	.		
152	)	001	
153	&		
154	+		
155	)		
156	P	002	001
157	P	004	001
158	P	003	001
159	.		
160	)	002	
161	,		
162	OUT		
163	END		
164	RQ		
165	)	005	
166	,		
167	END		
168	$		

```
                LIST      24
(               NUMBER                  ·1
)
                LI ST     61
(               LI ST     62
·(              NUMB&R                  2
                NUMBER                  2
·)
)
                LI ST     142
(               LI ST     143
·(              LI ST     145
··(             NUMBER                  3
                NUMBER                  3
                NUMBER                  3
··)
                LI ST     148
··(             NUMB&R                  3
                NUMB&R                  3
                NUMB&R                  3
··)
·)
)
                LI ST     353
(               LI ST     354
·(              LI ST     356
··(             LI ST     359
···(            NUMB&R                  4
                NUMB&R                  4
                NUMBER                  4
                NUMBER                  4
···)
                LI ST     363
···(            NUMB&R                  4
                NUMBER                  4
                NUMB&R                  4
                NUMB&R                  4
···)
                LI ST     367
···(            NUMBER                  4
                NUMBER                  4
                NUMBER                  a
                NUMBER                  4
··)
··)
                LI ST     371
·n(             LI ST     374
···(            NUMBER                  4
                NUMBER                  4
                NUMBER                  4
                NUMBER                  4
···)
                LI ST     378
···(            NUMBER                  4
                NUMBER                  4


                NUMB&H                  4
                NUMB&R                  4
···)
···(            LI ST     382
                NUMBER                  4
                NUMBER                  4
                NUMBER                  a
                NUMB&R                  a
···)
··)
·)
)
```

program

⊥ block ⊥

blockbody — statement — end

blockhead — blockbody — statement — ;

begin — blockhead — var decl — ; — blockhead — label decl — ;

new — λ — statement- — label — λ

expression — label def — statement-

expression- — λ — :

block — ifclause — truepart — expression-

goto — expression- — if — expression — then — expression — else

out — expression- — catena — variable — ← — expression-

disjunction — catena — & — primary

conjunction — disj head — disjunction

conjunction- — conjunction — ∨

negation — conj head — conjunction-

¬ — relation — relation — negation — ∧

choice — choice — rel op — choice

choice- — < — ≤ — = — ≠ — ≥ — >

sum — choice- — min — max — sum

sum-

+ — term — − — term — term — sum- — + — − — term

term~

factor

factor~

primary

term~ × / ÷ mod factor

factor~ ↑ primary

σ ln Ω

[expression]

tail primary

typetest variable

abs primary

isn isb isy isr isl isp isu isli

real primary

log val

length variable

integer primary

true false

reference

logical primary

proc def

@ variable

list primary

proc head expression '

variable list*

' proc head formal decl ,

list* formal λ

list head expression) list head)

variable

(list head expression , variable~

λ variable~ .

number variable~ [expression]

real*

real* 10

10 ± integer*

integer*

integer~ 10 integer* 10 ± integer*

digit integer~ digit integer* . integer*

0 1 2 3 4 5 6 7 8 9